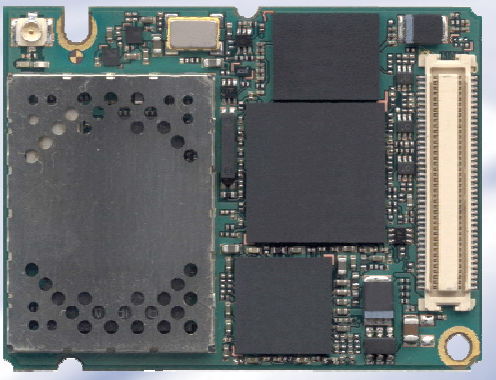
SIEMENS



JAVA™ Users Guide

TC65 JAVA User's Guide

Siemens Cellular Engine

Version: 05
DocID: TC65 JAVA User's Guide_V05

Document Name:	**TC65 JAVA User's Guide**
Version:	**05**
Date:	**September 26, 2005**
DocId:	**TC65 JAVA User's Guide_V05**
Status:	**Strictly confidential / Released**

General Notes
Product is deemed accepted by recipient and is provided without interface to recipient's products. The documentation and/or product are provided for testing, evaluation, integration and information purposes. The documentation and/or product are provided on an "as is" basis only and may contain deficiencies or inadequacies. The documentation and/or product are provided without warranty of any kind, express or implied. To the maximum extent permitted by applicable law, Siemens further disclaims all warranties, including without limitation any implied warranties of merchantability, completeness, fitness for a particular purpose and non-infringement of third-party rights. The entire risk arising out of the use or performance of the product and documentation remains with recipient. This product is not intended for use in life support appliances, devices or systems where a malfunction of the product can reasonably be expected to result in personal injury. Applications incorporating the described product must be designed to be in accordance with the technical specifications provided in these guidelines. Failure to comply with any of the required procedures can result in malfunctions or serious discrepancies in results. Furthermore, all safety instructions regarding the use of mobile technical systems, including GSM products, which also apply to cellular phones must be followed. Siemens or its suppliers shall, regardless of any legal theory upon which the claim is based, not be liable for any consequential, incidental, direct, indirect, punitive or other damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information or data, or other pecuniary loss) arising out the use of or inability to use the documentation and/or product, even if Siemens has been advised of the possibility of such damages. The foregoing limitations of liability shall not apply in case of mandatory liability, e.g. under the German Product Liability Act, in case of intent, gross negligence, injury of life, body or health, or breach of a condition which goes to the root of the contract. However, claims for damages arising from a breach of a condition, which goes to the root of the contract, shall be limited to the foreseeable damage, which is intrinsic to the contract, unless caused by intent or gross negligence or based on liability for injury of life, body or health. The above provision does not imply a change on the burden of proof to the detriment of the recipient. Subject to change without notice at any time. The interpretation of this general note shall be governed and construed according to German law without reference to any other substantive law.

Copyright
Transmittal, reproduction, dissemination and/or editing of this document as well as utilization of its contents and communication thereof to others without express authorization are prohibited. Offenders will be held liable for payment of damages. All rights created by patent grant or registration of a utility model or design patent are reserved.

Copyright © Siemens AG 2005

Trademark notices
MS Windows® is a registered trademark of Microsoft Corporation.
Java™ and Sun™ Java Studio Mobility 6 2004Q3 are registered trademarks of Sun Microsystems Inc.
Borland® JBuilder® is a registered trademark of Borland Software Corporation

Table of Contents

1 Preamble 8

2 Overview 9

2.1 Related Documents 9

2.2 Terms and Abbreviations 10

3 Installation 11

3.1 System Requirements 11

3.2 Installation CD 11
3.2.1 Components 12
3.2.1.1 Module Exchange Suite 12
3.2.1.2 WTK 12
3.2.1.3 SDK and Java Studio 12

3.3 Siemens Mobility Toolkit Installation 13
3.3.1 Installing the Standard Development Toolkit 13
3.3.2 Installing the SMTK Environment 13
3.3.3 Installing Sun Java Studio Mobility 6 14
3.3.4 Installing Eclipse 3.0 14
3.3.5 Installing Borland JBuilder X and 2005 14
3.3.6 Installing Module Exchange Suite (MES) 14

3.4 SMTK Uninstall 15

3.5 Upgrades 15

4 Software Platform 16

4.1 Software Architecture 16

4.2 Interfaces 17
4.2.1 ASC0 - Serial Device 17
4.2.2 General Purpose I/O 17
4.2.3 DAC/ADC 17
4.2.4 ASC1 17
4.2.5 Digital Audio Interface (DAI) 17
4.2.6 I2C/SPI 17
4.2.7 JVM Interfaces 18
4.2.7.1 IP Networking 18
4.2.7.2 Media 18
4.2.7.3 Other Interfaces 18

4.3 Data Flow of a Java Application Running on the Module 19

4.4 Handling Interfaces and Data Service Resources 20
4.4.1 Module States 20
4.4.1.1 State 1: Default – No Java Running 21
4.4.1.2 State 2: No Java Running, General Purpose I/O and I2C 21
4.4.1.3 State 3: No Java Running, General Purpose I/O and SPI 21
4.4.1.4 State 4: Default – Java Application Active 21
4.4.1.5 State 5: Java Application Active, General Purpose I/O and I2C 22
4.4.1.6 State 6: Java Application Active, General Purpose I/O and SPI 22
4.4.2 Module State Transitions 23

5 Maintenance 24

5.1 IP Service 24
5.2 Power Saving 25
5.3 Charging 25
5.4 Airplane Mode 26
5.5 Alarm 26
5.6 Shutdown 26
5.6.1 Automatic Shutdown 26
5.6.2 Manual Shutdown 27
5.6.3 Restart after Switch Off 27
5.7 Special AT Command Set for Java Applications 27
5.7.1 Switching from Data Mode to Command Mode 27
5.7.2 Mode Indication after MIDlet Startup 27
5.7.3 Long Responses 27
5.7.4 Configuration of Serial Interface 28
5.7.5 Java Commands 28
5.8 Restrictions 28
5.8.1 Flash File System 28
5.8.2 Memory 28
5.9 Performance 29
5.9.1 Java 29
5.9.2 Pin I/O 30
5.9.3 Data Rates on RS-232 API 30
5.9.3.1 Plain Serial Interface 31
5.9.3.2 Voice Call in Parallel 31
5.9.3.3 Scenarios with GPRS Connection 32
5.9.3.3.1 Upload 32
5.9.3.3.2 Download 33

6 MIDlets 34

6.1 MIDlet Documentation 34
6.2 MIDlet Life Cycle 34
6.3 Hello World MIDlet 36

7 File Transfer to Module 37

7.1 Module Exchange Suite 37
7.1.1 Windows Based 37
7.1.2 Command Line Based 37
7.2 Over the Air Provisioning 38
7.3 Security Issues 38
7.3.1 Module Exchange Suite 38
7.3.2 OTAP 38

8 Over The Air Provisioning (OTAP) 39

8.1 Introduction to OTAP 39
8.2 OTAP Overview 39
8.3 OTAP Parameters 40

8.4 Short Message Format 41
8.5 Java File Format 42
8.6 Procedures 43
8.6.1 Install/Update 43
8.6.2 Delete 44
8.7 Time Out Values and Result Codes 45
8.8 Tips and Tricks for OTAP 45
8.9 OTAP Tracer 46
8.10 Security 46
8.11 How To 47

9 Compile and Run a Program without a Java IDE 48

9.1 Build Results 48
9.2 Compile 49
9.3 Run on the Module with Manual Start 49
9.4 Run on the Module with Autostart 49
9.4.1 Switch on Autostart 50
9.4.2 Switch off Autostart 50

10 Debug Environment 51

10.1 Data Flow of a Java Application in the Debug Environment 51
10.2 Emulator 52
10.3 Java IDE 53
10.3.1 Sun Java Studio Mobility 6 2004Q3 54
10.3.1.1 Switching emulators 55
10.3.1.2 Projects 56
10.3.1.3 Templates 57
10.3.1.4 Examples 57
10.3.1.5 Compile and run 57
10.3.2 Borland JBuilder X 58
10.3.2.1 Examples 59
10.3.3 Borland JBuilder 2005 61
10.3.3.1 Examples 62
10.3.4 Eclipse 3.0 62
10.3.4.1 Integration 62
10.3.4.2 Switching Emulators 64
10.3.4.3 Example 65
10.3.4.4 Compile and debug 67
10.4 Breakpoints 68

11 Java Security 69

11.1 Secure Data Transfer 70
11.1.1 Create a Secure Data Transfer Environment Step by Step 72
11.2 Execution Control 73
11.2.1 Change to Secured Mode Concept 74
11.2.2 Concept for the Signing the Java MIDlet 75
11.3 Application and Data Protection 76

11.4 Structure and Description of the Java Security Commands 76
11.4.1 Structure of the Java Security Commands 77
11.4.2 Build Java Security Command 78
11.4.3 Send Java Security Command to the Module 79

11.5 Create a Java Security Environment Step by Step 80
11.5.1 Create Key store 80
11.5.2 Export X.509 Root Certificate 80
11.5.3 Create Java Security Commands 80
11.5.4 Sign a MIDlet 81

11.6 Attention 81

12 Java Tutorial 82

12.1 Using the AT Command API 82
12.1.1 Class ATCommand 82
12.1.1.1 Instantiation with or without CSD Support 82
12.1.1.2 Sending an AT Command to the Device, the send() Method 83
12.1.1.3 Data Connections 83
12.1.1.4 Synchronization 85
12.1.2 ATCommandResponseListener Interface 85
12.1.2.1 Non-blocking ATCommand.send() Method 85
12.1.3 ATCommandListener Interface 86
12.1.3.1 ATEvents 86
12.1.3.2 Implementation 86
12.1.3.3 Registering a Listener with an ATCommand Instance 87

12.2 Programming the MIDlet 88
12.2.1 Threads 88
12.2.2 Example 88

13 Differences from the TC45 90

Figures

Figure 1: Overview 9
Figure 2: Interface Configuration 18
Figure 3: Data flow of a Java application running on the module 19
Figure 4: Module State 1 21
Figure 5: Module State 2 21
Figure 6: Module State 3 21
Figure 7: Module State 4 21
Figure 8: Module State 5 22
Figure 9: Module State 6 22
Figure 10: Module State Transition Diagram 23
Figure 11: Test case for measuring Java command execution throughput 29
Figure 12: Test case for measuring Java MIDlet performance and handling pin-IO 30
Figure 13: Scenario for testing data rates on ASC1 31
Figure 14: Scenario for testing data rates on ASC1 with a voice call in parallel 31
Figure 15: Scenario for testing data rates on ASC1 with GPRS data upload 32
Figure 16: Scenario for testing data rates on ASC1 with GPRS data download 33
Figure 17: OTAP Overview 39
Figure 18: OTAP: Install/Update Information Flow 43
Figure 19: OTAP: Delete Information Flow 44
Figure 20: Data flow of a Java application in the debug environment 51
Figure 21: Sun Java Studio Mobility 6 - The installed emulators 54

Figure 22: Sun Java Studio Mobility 6 - Switching Emulators 55
Figure 23: Sun Java Studio Mobility 6 - Project Manager 56
Figure 24: Sun Java Studio Mobility 6 - Selecting a template 57
Figure 25: JBuilderX – JDK settings 58
Figure 26: JBuilderX – Siemens Library 58
Figure 27: JBuilderX – Sample Projects 59
Figure 28: JBuilderX – Starting the debugging session 60
Figure 29: JBuilder2005 – JDK settings 61
Figure 30: JBuilderX – Siemens Library 61
Figure 31: JBuilder2005 – Sample Projects 62
Figure 32: Eclipse – Plug-in installation 63
Figure 33: Eclipse – Plug-in installation, restart 63
Figure 34: Eclipse – IMP-NG component 64
Figure 35: Eclipse – J2ME platform 64
Figure 36: Eclipse – Project import 65
Figure 37: Eclipse - Example 66
Figure 38: Eclipse – Create package 67
Figure 39: Eclipse - Configuration 68
Figure 40: Mode 1 - Java Security not activated 70
Figure 41: Mode 2 - Java Security activated (server certificate = certificate into module) 71
Figure 42: Mode 2 - Java Security activated (server certificate and self signed root certificate in module form a chain) 71
Figure 43: Switch to Security Mode 74
Figure 44: Prepare MIDlet for Secured Mode 75
Figure 45: Build Java Security Command 78

Tables

Table 1: Download data rate with different number of timeslots, CS2 32
Table 2: Download data rate with different number of timeslots, CS4 32
Table 3: Download data rate with different number of timeslots, CS2 33
Table 4: Download data rate with different number of timeslots, CS4 33
Table 5: A typical sequence of MIDlet execution 35
Table 6: Parameters and keywords 40

1 Preamble

This document is also valid for the TC65 Terminal with the main exception that the terminal does not feature the USB, ASC1, DAC and DAI interface. For other exceptions and differences please see [3] and [4].

2 Overview

The TC65 module features an ultra-low profile and low-power consumption for data (CSD and GPRS), voice, SMS and fax. Java technology and several peripheral interfaces on the module allow you to easily integrate your application.

This document explains how to work with the TC65 module, the installation CD and the tools provided on the installation CD.

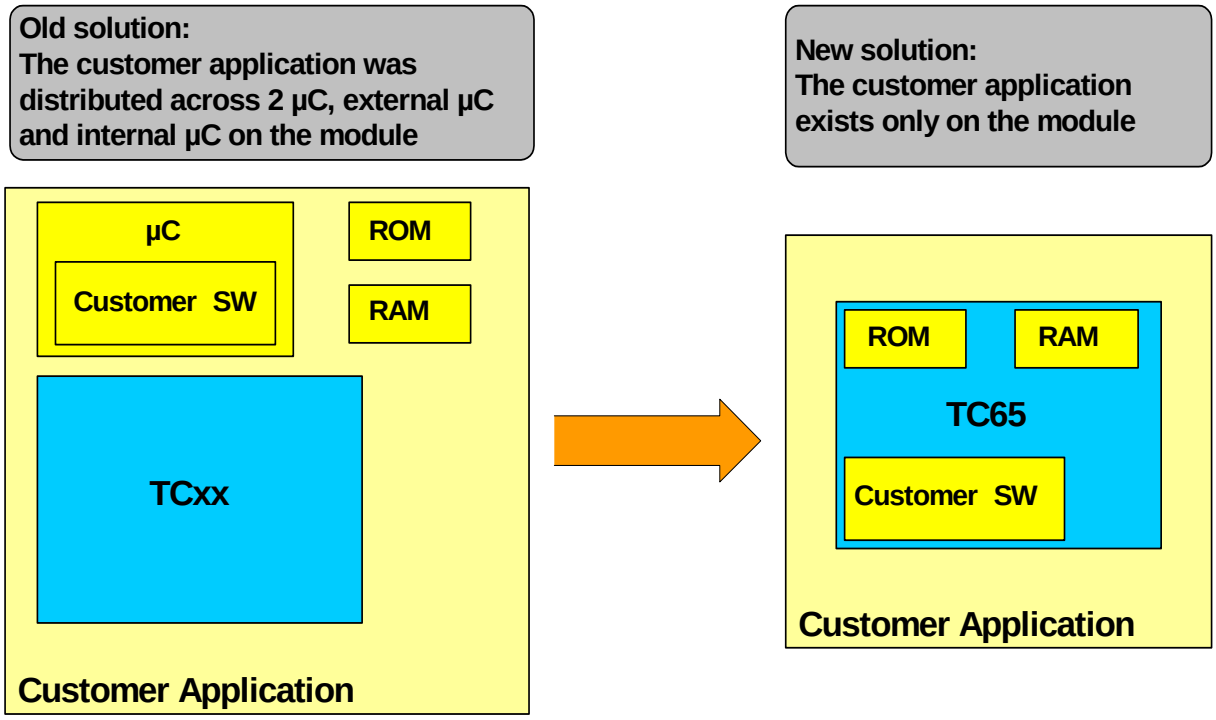


Figure 1: Overview

2.1 Related Documents

In addition to the Java Docs for the development API (see Chapter 4), the following documents are included with the SMTK:

[1] Multiplexer Installation Guide
[2] DSB75 Support Box - Evaluation Kit for Siemens Cellular Engines
[3] TC65 AT Command Set
[4] TC65 Hardware Interface Description
[5] Java doc \wtk\doc\html\index.html
[6] IMP-NG, JSR228, Standard
[7] Application Note 24: Application Developer's Guide

2.2 Terms and Abbreviations

Abbreviation	Description
API	Application Program Interface
ASC	Asynchronous Serial Controller
CLDC	Connected Limited Device Configuration
CSD	Circuit-Switched Data
DAI	Digital Audio Interface
DCD	Data Carrier Detect
DSR	Data Set Ready
GPIO	General Purpose I/O
GPRS	General Packet Radio Service
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IP	Internet Protocol
J2ME™	Java 2 Mobile Edition
J2SE™	Java 2 Standard Edition
JAD	Java Application Description
JAR	Java Archive
JDK	Java Development Kit
JVM	Java Virtual Machine
LED	Light Emitting Diode
ME	Mobile Engine
MIDP	Mobile Information Device Protocol
OTA	Over The Air
OTAP	Over The Air Provisioning of Java Applications
PDP	Packet Data Protocol
PDU	Protocol Data Unit
SDK	Standard Development Kit
SMS	Short Message Service
SMTK	Siemens Mobile Toolkit
TCP	Transfer Control Protocol
URC	Unsolicited Result Code
URL	Universal Resource Locator
VBS	Visual Basic Script
WTK	Wireless Toolkit

3 Installation

3.1 System Requirements

The Siemens Mobility Toolkit (SMTK) TC65 requires that you have:
1. Windows 2000 or Windows XP installed
2. 40Mbytes free disk space for SMTK
3. Administration privileges
4. Java 2 SDK, Standard Edition 1.4. To install the JDK version 1.4.2_07 provided, follow the instructions in Section 3.3.1.

If a Java IDE such as Sun Java Studio Mobility 6 2004Q3, Eclipse 3.0.1, Eclipse 3.0.2, JBuilder X or 2005 is installed, it can be integrated into the SMTK environment during the installation of the SMTK. To install one of the IDEs, follow the installation instructions in Section 3.3.3 and Section 3.3.4 respectively.

3.2 Installation CD

The Siemens Mobility Toolkit TC65 Installation CD includes:
- Module Exchange Suite
- EclipseME plugin
- TC65 WTK
  bin
  - various tools
  doc
  - html
  - java docs for APIs
  lib
  - classes.zip
  src
  - various examples
- Java SDK
  J2sdk-1_4_2_07-windows-i586-p.exe
- Sun Java Studio Mobility 6
  jstudio_M04q3-win-ml.exe
- Documents:
  DSB75_HW_Description.pdf
  TC65_AT_Command_Set.pdf
  TC65_HW_Description.pdf
  TC65T_HW_Description.pdf
  TC65_ReleaseNote.pdf
  WM_AN_24_Dev_Guide.pdf
  TC65_Java_UserGuide.pdf (this document)

Some of the content can only be accessed after the installation.

3.2.1 Components

3.2.1.1 Module Exchange Suite

The Module Exchange Suite allows the developer to access the Flash file system on the cellular engine from the development environment over a serial interface. File transfers from PC to module are greatly facilitated by this suite.

3.2.1.2 WTK

wtk is the directory where all the necessary components for TC65 Java application creation and debugging are stored.

3.2.1.3 SDK and Java Studio

This is software provided by SUN to support Java application development.

3.3 Siemens Mobility Toolkit Installation

The SMTK comes with an installation CD. The installation program automatically installs the necessary components and IDE integrations. Software can be uninstalled and updated with the install program. The next sections cover the installation and removal of the SMTK and the installation of the SDK and the supported IDEs.

3.3.1 Installing the Standard Development Toolkit

1. The JDK version 1.4.2_07 is provided on the TC65 SMTK installation disk in the subdirectory “JDK 1.4”. To begin the installation, start the j2sdk-1_4_2_07-windows-i586-p.exe and follow the instructions of the JDK setup procedure. If there is no JDK installed on the target machine the installation of the provided JDK will be offered automatically during the SMTK installation process.
2. Once the toolkit has been installed, the environment variable “path” can be altered to comfortably use the JDK tools. This is not necessary for using the Siemens SMTK.
3. Open the Control Panel.
   a) Open *System.*
   b) Click on *Advanced.*
   c) Click on the *Environment Variables* button.
   d) Choose *path* from the list of system variables.
   e) Append the path for the bin directory of the newly installed SDK to the list of directories for the *path* variable.

3.3.2 Installing the SMTK Environment

Before you start the installation please make sure all applications, especially the IDEs are closed.

1. Insert CD, start setup.exe. When the dialog box appears simply press the “Next” button to continue the procedure.
2. You will be asked to read the license agreement. If you accept the agreement, press “Yes” to continue with the installation.
3. A file including special information about the installation and use of the SMTK is shown. Press “Next” to continue.
4. You will be asked to enter the path name where Eclipse 3.0.1 or 3.0.2 is installed. Please type in the folder where Eclipse with the ME plugin is installed and press “Next”. If you have not installed Eclipse or do not want to integrate the SMTK into Eclipse, please press “Next” without typing in a selected folder.
5. The installation software checks for the Java SDK. If there is no SDK on the system the installation procedure now offers to install the provided JDK. If this step is refused, the setup process will not continue because a properly installed JDK is mandatory for using the SMTK environment.
6. At this point, the installation software checks for a Java IDE to be integrated with the SMTK. A Java IDE is not necessary to use the TC65 SMTK. The IDE installation can be done at any time even if the TC65 SMTK is already installed. To integrate the SMTK into the Java IDE run the SMTK setup program in maintenance mode again. However, you can continue the setup procedure and install the IDE installation later or cancel the setup program at this stage and restart it after installing one of the supported Java IDEs. In case you wish to install a Java IDE please follow the instructions below and in Section 3.3.3. If no installed IDE is found the TC65 SMTK offers to install SUN Java Studio Mobility 6 2004Q3. Alternatively, you can install the SUN IDE by following the instructions in Section 3.3.3.

7. If the SDK and one or more Java IDEs are found, you will be asked to choose which IDE you want integrated into the TC65 development environment. Once an IDE has been found and selected, press "Next" to continue. Ensure that your Java IDE is closed.
8. Select the folder where the TC65 SMTK will be installed. A folder will be suggested to you but you may browse to select a different one.
9. Choose the path that TC65 will appear under in the Start Menu.
10. A brief summary of all entries made is shown. Press “Next” to continue.
11. A dialog box will inform you that the Module Exchange Suite (MES) will be installed in the next step. Please press “OK”. A separate setup wizard for the Module Exchange Suite will be opened. Please follow the setup wizard’s instructions.
12. After step 11, all necessary files will be copied from the CD into the target folder.
13. This is the final step. Again, a listing of all installed components appears. Please press “Finish” to end the installation

3.3.3 Installing Sun Java Studio Mobility 6

1. Sun Java Studio Mobility 6 is provided on the TC65 SMTK installation disk in the subdirectory “SJSM6”. To begin installation, start jstudio_M04q3-win-ml.exe and follow the Sun Studio setup procedure instructions.
2. On the first use of Sun™ Studio 6 after installation, you will be prompted to specify a personal Java folder. Each user may have their own Java folder.

   Note: The integration of the SMTK into Sun™ Studio 6 is only possible if the personal user folder is set. It can only be rolled back by the user who installed the SMTK. If all users use the same Java folder, any user may roll back the integration.

3.3.4 Installing Eclipse 3.0

Eclipse can be freely downloaded from http://www.eclipse.org. In order to use Eclipse with the TC65 the EclipseME plug-in is also needed. It can be downloaded from http://eclipseme.org/. A customized version of this plug-in also comes with TC65 SMTK. It is currently recommend that this version be used.

3.3.5 Installing Borland JBuilder X and 2005

Borland JBuilder can be purchased from http://www.borland.com/jbuilder.

Note: The installation path name of JBuilder should not include space characters.

There are also 30 days trial versions available on the website. Installation instructions can be found on the web page.

3.3.6 Installing Module Exchange Suite (MES)

The Module Exchange Suite (MES) is installed during the SMTK installation. If you would like to install the Module Exchange Suite separately, repair or remove it, please use the Module Exchange Suite (MES) setup.exe, which is located on the TC65 SMTK installation disk in the subdirectory “MES”.

3.4 SMTK Uninstall

The TC65 SMTK install package comes with an uninstall facility. The entire SMTK or parts of the package can be removed. To start the uninstall facility, open the *Control Panel*, select *Add/Remove Programs*, select *TC65 Software Development Kit* and follow the instructions.

The Module Exchange Suite (MES) is not uninstalled automatically with the SMTK. If you would like to uninstall the Module Exchange Suite (MES) as well, please run the MES uninstall facility. To run the uninstall program, open the *Control Panel*, select *Add/Remove Programs*, select *Siemens Module Exchange Suite (MES)* and follow the instructions.

3.5 Upgrades

The SMTK can be modified, repaired or removed by running the setup program on the Installation CD.

4 Software Platform

In this chapter, we discuss the software architecture of the SMTK and the interfaces to it.

4.1 Software Architecture

The SMTK enables a customer to develop a Java application on a PC and have it be executable on the TC65 module. The application is then loaded onto the module. The platform is comprised of:

- the Java™ 2 Micro Edition (J2ME™), which forms the base of the architecture.
  The J2ME™ is provided by SUN Microsystems, http://java.sun.com/j2me/. It is specifically designed for embedded systems and has a small memory footprint. TC65 uses:

  CLDC 1.1 HI, the connected limited device configuration hot spot implementation.

  IMP-NG, the information module profile 2nd generation, this is for the most part identical to MIDP 2.0 but without the lcdui package.

- Additional Java virtual machine interfaces:
  AT Command API
  File I/O API

  The data flow through these interfaces is shown in Figure 3 and Figure 20.

- Memory space for Java programs:
  Flash File System: around 1700k
  RAM: around 400k

  Application code and data share the space in the flash file system and in RAM.

- Additional accessible periphery for Java applications
  - A maximum of ten digital I/O pins usable, for example, as:
    Output: status LEDs
  - Input: Emergency Button
  - One I2C/SPI Interface.
  - One Digital Analog Converter and two Analog Digital Converters.
  - Serial interface (RS-232 API): This standard serial interface could be used, for example, with an external GPS device or a current meter.
    For detailed information see chapter 4.2.

4.2 Interfaces

4.2.1 ASC0 - Serial Device

ASC0, an Asynchronous Serial Controller, is a 9-wire serial interface. It is described in the Hardware Interface Description [4]. Without a running Java application the module can be controlled by sending AT commands over ASC0. Furthermore, ASC0 is designed for transferring files from the development PC to the module and for controlling the module with AT commands. When a Java application is started, ASC0 can be used as an RS-232 port. Refer to the Java doc [5] for details.

4.2.2 General Purpose I/O

There are ten I/O pins that can be configured for general purpose I/O. When the TC65 starts up, all 10 pins are set, by default, to a high-impedance state for use as input. One pin can be configured as a pulse counter. All lines can be accessed under Java by AT commands. See [3] and [4] for information on configuring the pins.

4.2.3 DAC/ADC

There are two analogue input lines and one analogue output line. They are accessed by AT commands. See [3] and [4] for details.

4.2.4 ASC1

ASC1 is the second serial interface on the module. This is a 4-pin interface (RX, TX, RTS, CTS). It can be used as a second AT interface when a Java application is not running or by a running Java application as System.out.

4.2.5 Digital Audio Interface (DAI)

The TC65 has a seven-line serial interface with one input data clock line and input/output data and frame lines to support the DAI. Refer to the AT Command Set [3] and Hardware Interface Description [4] documents for more information.

4.2.6 I2C/SPI

There is a 4 line serial interface which can be used as I2C or SPI interface. It is described in the Hardware Interface Description [4]. The at^sspi at command configures and drives this interface. For details see [4].

4.2.7 JVM Interfaces

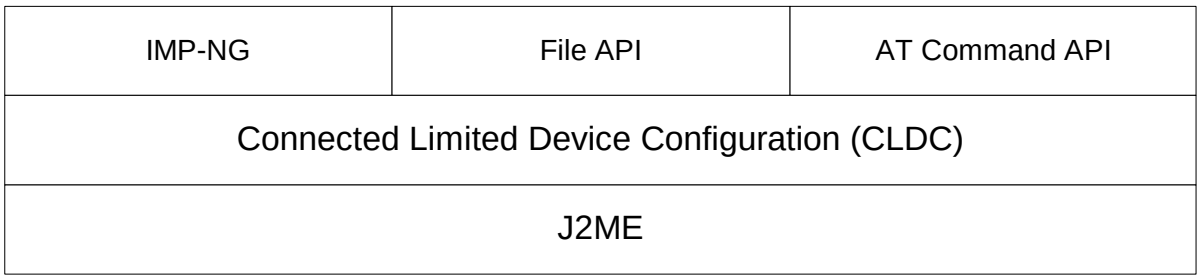


Figure 2: Interface Configuration

J2ME, CLDC and MIDP were implemented by SUN. IMP-NG is a stripped down version of MIDP 2.0 prepared by Siemens and does not include the graphical interface LCDUI. Siemens developed the File I/O API and the AT command API. Documentation for J2ME and CLDC can be found at http://java.sun.com/j2me/. Documentation for the other APIs is found in …/ Java doc [5].

4.2.7.1 IP Networking

IMP-NG provides access to TCP/IP similarly to MIDP 2.0.

Because the used network connection, CSD or GPRS, is fully transparent to the Java interface, the CSD and GPRS parameters must be defined separately either by the AT command at^sjnet [3] or by parameters given to the connector open method, see Java doc [5].

4.2.7.2 Media

TC65 does not support the media package. See Java doc [5].

4.2.7.3 Other Interfaces

TC65 supports neither the PushRegistry interfaces and mechanisms nor any URL schemes for the PlatformRequest method. See Java doc [5].

4.3 Data Flow of a Java Application Running on the Module

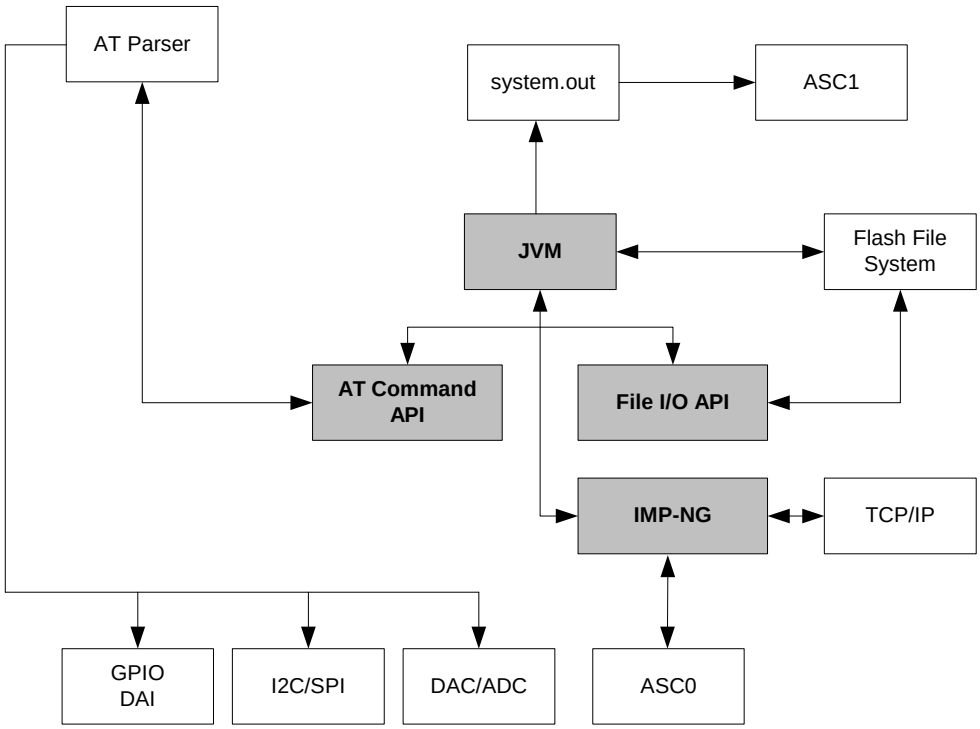


Figure 3: Data flow of a Java application running on the module.

The diagram shows the data flow of a Java application running on the module. The data flow of a Java application running in the debug environment can be found in Figure 20.

The compiled Java applications are stored as JAR files in the Flash File System of module. When the application is started, the JVM interprets the JAR file and calls the interfaces to the module environment.

The module environment consists of the:

- Flash File System: available memory for Java applications
- TCP/IP: module internal TCP/IP stack
- GPIO: general purpose I/O
- DAI: Digital Audio Interface
- ASC0: Asynchronous serial interface 0
- ASC1: Asynchronous serial interface 1
- I2C: I2C bus interface
- SPI: Serial Peripheral Interface
- DAC: digital analog converter
- ADC: analog digital converter
- AT parser: accessible AT parser

The Java environment on the module consists of the:

- JVM: Java Virtual Machine
- AT command API: Java API to AT parser
- File API: Java API to Flash File System
- IMP-NG: Java API to TCP/IP and ASC0

4.4 Handling Interfaces and Data Service Resources

To develop Java applications the developer must know which resources, data services and hardware access are available.

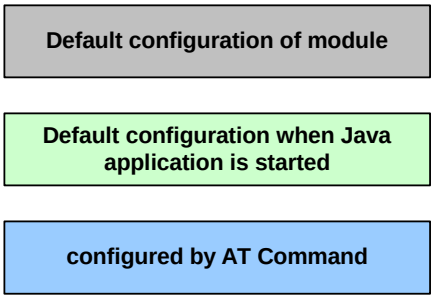
- There are three AT parsers available
- There is hardware access over
  - two serial interfaces: ASC1 (System.out only) and ASC0 (fully accessible).
  - general purpose I/O. To configure the hardware access, please refer to the AT Command Set [3] and the Hardware Interface Description [4].
  - I2C/SPI
  - All restrictions of combinations are described in section 4.4.1.

- A Java application has:
  - three instances of the AT command class, one with CSD and two without, each of which would, in turn, be attached to one of the three AT parsers.
  - one instance of access to a serial interface, ASC0, through the RS-232 API.
  - System.out over the serial interface, ASC1, for debugging.

4.4.1 Module States

The module can exist in the following six states in relation to a Java application, the serial interfaces, GPIO and I2C/SPI. See the AT Command Set [3] for information about any AT commands referenced. A state transition diagram is shown in
Figure 10.

This section shows how Java applications must share AT parsers, GPIO pins and I2C/SPI resources. DAC, ADC and DAI are not discussed here. The USB interface is an alternative to ASC1. When the USB is plugged in, the ASC1 interface is deactivated.



Color legend for the following figures

4.4.1.1 State 1: Default – No Java Running

This is the default state. The Java application is inactive and there is an AT interface with CSD on ASC0 as well as ASC1. All HW interface pins are configured as inputs.

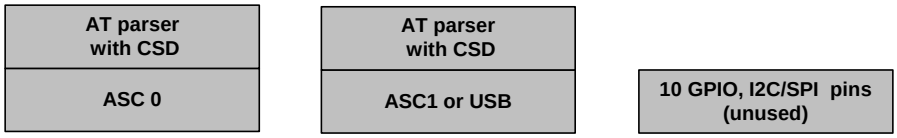


Figure 4: Module State 1

4.4.1.2 State 2: No Java Running, General Purpose I/O and I2C

The Java application is inactive. There is an AT parser with CSD on ASC0 as well as ASC1. Up to ten I/0 pins are used as general purpose I/O plus a I2C interface. The pins are configured *by* at^scpin or at^sspi (refer *to* AT Command Set [3]).

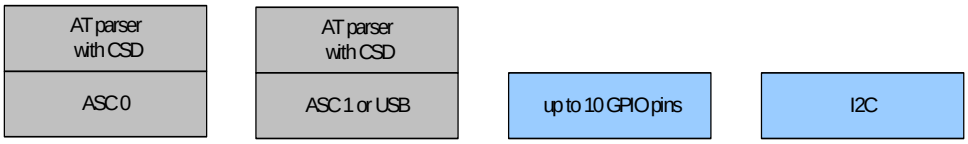


Figure 5: Module State 2

4.4.1.3 State 3: No Java Running, General Purpose I/O and SPI

The Java application is inactive and there is an AT interface with CSD on ASC0 as well as ASC1. There is an SPI interface as well at ten I/0 pins that can be used for general purpose I/O. The pins are configured with at^scpin or at^sspi (refer to AT Command Set [3]).

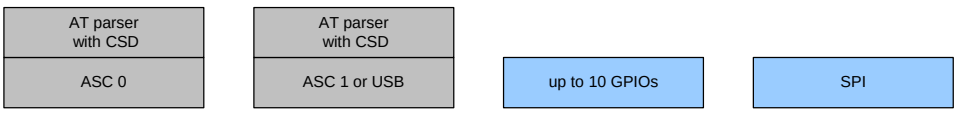


Figure 6: Module State 3

4.4.1.4 State 4: Default – Java Application Active

The Java application is active and ASC1 is used as System.out and the Java instance of the RS-232 serial interface is connected to ASC0. Java instances of AT commands are connected to the available AT parsers. The Java application is activated with *at^sjra* (refer to AT Command Set [3]) or autostart.

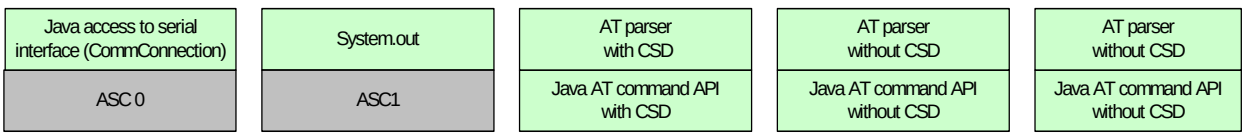

Java access to serial interface (CommConnection)
ASC 0
System.out
ASC1
AT parser with CSD
Java AT command API with CSD
AT parser without CSD
Java AT command API without CSD
AT parser without CSD
Java AT command API without CSD

Figure 7: Module State 4

4.4.1.5 State 5: Java Application Active, General Purpose I/O and I2C

The Java application is active, ASC1 is used as System.out and the Java instance of the RS-232 serial interface is connected to ASC0. The Java application is activated with *at^sjra.* The I/O pins are configured with at^scpin or at^sspi. Refer to the AT Command Set [3] for AT command details.

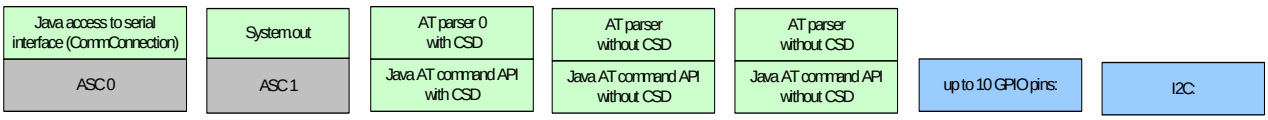

Java access to serial interface (CommConnection)
ASC 0
System.out
ASC 1
AT parser 0 with CSD
Java AT command API with CSD
AT parser without CSD
Java AT command API without CSD
AT parser without CSD
Java AT command API without CSD
up to 10 GPIO pins:
I2C

Figure 8: Module State 5

4.4.1.6 State 6: Java Application Active, General Purpose I/O and SPI

The Java application is running, ASC0 is used as System.out and the Java instance of the RS-232 serial interface is connected to ASC1. The Java application is activated with *at^sjra* (refer to the AT Command Set [3]).

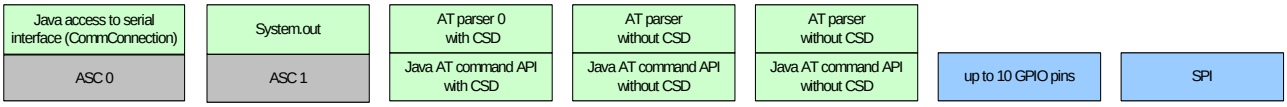

Java access to serial interface (CommConnection)
ASC 0
System.out
ASC 1
AT parser 0 with CSD
Java AT command API with CSD
AT parser without CSD
Java AT command API without CSD
AT parser without CSD
Java AT command API without CSD
up to 10 GPIO pins
SPI

Figure 9: Module State 6

4.4.2 Module State Transitions

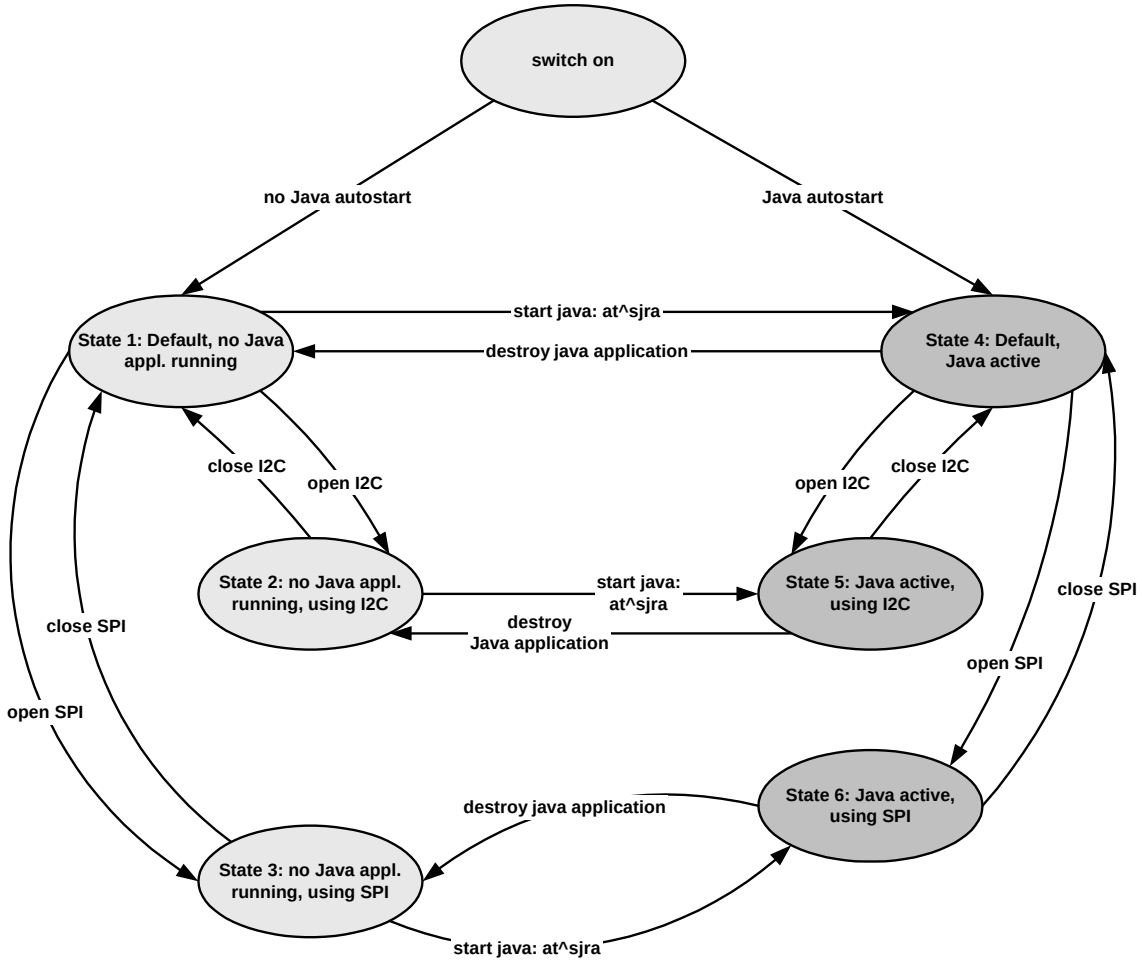


Figure 10: Module State Transition Diagram

Note: No AT parser is available over serial interface ASC0 or ASC1 while a Java application is running on the module. System.out is available on ASC1 for debugging while a Java application is running.

5 Maintenance

The basic maintenance features of the TC65 are described below. Explicit details of these functions and modes can be found in the AT Command Set [3] and the Hardware Interface Description [4].

5.1 IP Service

Apart from the standard Java IP networking interfaces (UDPDatagramConnection, SocketConnection, ...) the TC65 also supports a set of Internet Services via AT command. There are some correlations between the Java and the AT IP Services.

1. the connection profile 0 is also used by Java: when Java starts up a networking connection it tries to set and activate connection profile 0 with the parameters configured by at^sjnet or in the connector.open method.
2. Java tries to (re-)use an active Internet Service profile: if using connection profile 0 fails, because e.g. this (or another) connection profile is already used by the Internet Services, Java networking also uses this, already active, profile.
3. deactivation of the connection profile happens when all applications are “finished”: Java has its networking idle time, the Internet Services have the inactTO.

So that means that Java networking and AT Internet Services can be used in parallel but care has to be taken about configuring and activation of the connection profile. In the simplest case use connection profile 0 for the Internet Services and set the parameters to the same values as the Java networking parameters. This way it makes no difference whether the connection is activated by the Internet Services or Java.

There are some aspects which have to be kept in mind for all IP Services (Java and AT command):

- when an open TCP connection is cut (e.g. the other side dies/is switched off) it takes around 10 minutes during which retransmissions are send, until the situation is detected as an error (in Java an exception is thrown).
- the number of IP services used in parallel should be kept small. An active IP service uses up resources and may deteriorate the overall performance.

5.2 Power Saving

The module supports several power saving modes which can be configured by the AT command *at+cfun* [3]. Power saving affects the Java application in two ways. First, it limits access to the serial interface (RS-232-API) and the GPIO pins. Second, power saving efficiency is directly influenced by the way a Java application is programmed.

Java hardware access limitations:

- In NON-CYCLIC SLEEP mode (cfun=0) the serial interface cannot be accessed while in CYCLIC SLEEP mode (CFUN=7 or 9) the serial interface can be used with hardware flow control (CTS/RTS).
- In all SLEEP modes the GPIO polling frequency is reduced so that only signal changes which are less than 0.2Hz can be detected properly. Furthermore, the signal must be constant for at least 2.12s to detect any changes. For further details refer to [4].

Java power saving efficiency:

- As long as any Java thread is active, power consumption cannot be reduced, regardless whether any SLEEP mode has been activated or not. A Java application designed to be power efficient should not have any unnecessarily active threads (e.g. no busy loops).

5.3 Charging

Please refer to [3] and [4] for general information about charging. Charging can be monitored by the running Java application. The JVM is active in Charge mode and in Charge-Only mode if autostart is activated. Only a limited number of AT commands are available when the module is in Charge-Only mode. A Java application must be able to handle the Charge-Only mode and reset the module to reinstate the normal mode. See [4] for information about the Charge-Only mode.

The Charge-Only mode is indicated by URC “^SYSSTART CHARGE-ONLY MODE”.

Note: When a Java application is started in Charge-Only mode only AT Command APIs without CSD are available. The mode-indicating URC is created after issuing the very first AT command on any opened channel. To read the URC it is necessary to register a listener (see [5]) on this AT command API instance before passing the first AT command.

5.4 Airplane Mode

The main characteristic of this mode is that the RF is switched off and therefore only a limited set of AT commands is available. The mode can be entered or left using the appropriate at^scfg command. This AT command can also be used to configure the airplane mode as the standard startup mode, see [4]. The JVM is started when autostart is enabled. A Java application must be able to handle this mode. The airplane mode is indicated by URC "SYSSTART AIRPLANE MODE". Since the radio is off all classes related to networking connections, e.g. SocketConnection, UDPDatagramConnection, SocketServerConnection, HTTPConnection, will throw an exception when accessed.

5.5 Alarm

The ALARM can be set with the *at+cala* AT command. Please refer to the AT Command Set [3] and Hardware Interface Description [4] for more information. One can set an alarm, switch off the module with *at^smso*, and have the module restart at the time set with *at+cala*. When the alarm triggers the module restarts in a limited functionality mode, the "airplane mode". Only a limited number of AT commands are available in this mode, although the JVM is started when autostart is enabled. A Java application must be able to handle this mode and reset the module to reinstate the normal mode. The mode of a module started by an alarm is indicated by the URC "^SYSSTART AIRPLANE MODE".

Note: For detailed information which functionality is available in this mode see [4]. The mode indicating URC is created after issuing the very first AT command on any opened channel.

5.6 Shutdown

If an unexpected shutdown occurs, data scheduled to be written will get lost due to a buffered write access to the flash file system. The best and safest approach to powering down the module is to issue the AT^SMSO command. This procedure lets the engine log off from the network and allows the software to enter a secure state and save all data. Further details can be found in [4].

5.6.1 Automatic Shutdown

The module is switched off automatically in different situations:

- under- or overtemperature
- under- or overvoltage

The shutdown will happen without a warning notification unless the appropriate URC has been activated. If the URCs are enabled, the module will deliver an alert before switching off. To activate the URCs for temperature conditions use the *at^sctm* command; to activate the voltage condition URCs use the *at^sbc* command. It is recommended that these URCs be activated so that the module can be shut by the application with *at^smso* after setting an alarm, see Section 5.5. The commands are described in the AT Command Set [3], while a description of the shutdown procedure can be found in [4].

5.6.2 Manual Shutdown

The module can be switched off manually with the AT command, at^smso or when using the TC65 terminal by pressing the ignition key for a period of time (see [4]). In these cases the midlets destroyApp method is called and the application has 5s time to clean up and call the notifydestroy method. After the 5s the VM is shut down.

5.6.3 Restart after Switch Off

When the module is switched off without setting an alarm time (see the AT Command Set [3]), e.g. after a power failure, external hardware must restart the module with the Ignition line (IGT). The Hardware Interface Description [4] explains how to handle a switched off situation.

5.7 Special AT Command Set for Java Applications

For the full AT command set refer to [3]. There are differences in the behaviour AT commands issued from a Java application in comparison to AT commands issued over a serial interface.

5.7.1 Switching from Data Mode to Command Mode

Cancellation of the data flow with "+++" is not available in Java applications, see [3] for details. To break the data flow use *breakConnection().* Refer to \wtk\doc\index.html [5].

5.7.2 Mode Indication after MIDlet Startup

After starting a module without autobauding on, the startup state is indicated over the serial interface. Similarly, after MIDlet startup the module sends its startup state (^SYSSTART, ^SYSSTART ALARM MODE etc.) to the MIDlet. This is done via a URC to the AT Command API instance which executes the very first AT Command from within Java. To read this URC it is necessary to register a listener (see [5]) on this AT Command API instance before passing the first AT Command.

5.7.3 Long Responses

The AT Command API can handle responses of AT commands up to a length of 1024 bytes. Some AT commands have responses longer than 1024 bytes, for these responses the Java application will receive an Exception.

Existing workarounds:

- Instead of listing the whole phone book, read the entries one by one
- Instead of listing the entire short message memory, again list message after message
- Similarly, read the provider list piecewise
- Periods of monitoring commands have to be handled by Java, e.g. *at^moni, at^smong*. These AT commands have to be used without parameters, e.g. for *at^moni* the periods must be implemented in Java.

5.7.4 Configuration of Serial Interface

While a Java application is running on the module, only the AT Command API is able to handle AT commands. All AT commands referring to a serial interface are ignored. In particular these commands:

- AT+IPR
- AT\Q3

If Java is running, the firmware will ignore any settings from these commands. Responses to the read, write or test commands will be invalid or deliver “ERROR”.

Note: When a Java application is running, all settings of the serial interface are done with the class CommConnection. This is fully independent of any AT commands relating to a serial interface.

5.7.5 Java Commands

There is a small set of special Java AT commands:

- at^sjra, start a Java application
- at^sjnet, configuration of Java networking connections
- at^sjotap, start and configuration of over the air provisioning
- at^sjsec, security configuration

Refer to the AT command set [3].

5.8 Restrictions

5.8.1 Flash File System

The maximum length of a complete path name, including the path and the filename, is limited by the Flash file system on the module to 124 characters. It is recommended names of classes and files be distinguished by more than case sensitivity.

5.8.2 Memory

The CLDC 1.1 HI features a just-in-time compiler. This means that parts of the Java byte code which are frequently executed are translated into machine code to improve efficiency. This feature uses up RAM space. There is always a trade off between code translation to speed up execution and RAM space available for the application.

5.9 Performance

The performance study was focused on comparable performance values under various circumstances.

5.9.1 Java

This section gives information about the Java command execution throughput ("jPS"= Java statements per second). The scope of this measurement is only the statement execution time, not the execution delay (Java command on AT interface ➔ Java instruction execution ➔ reaction on GPIO).

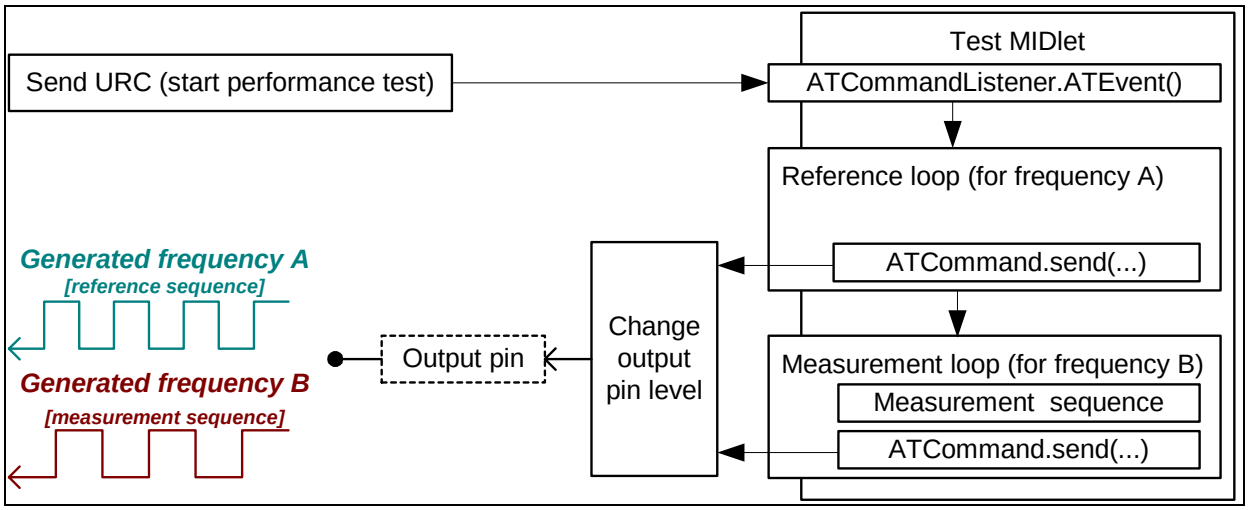


Figure 11: Test case for measuring Java command execution throughput

The following Java instruction was used for calculation of the typical jPS:

value = (2 x number of calculation statements) /
((1 / frequencyB) - (1 / frequencyA));

Measurement and calculation were done using:

- **duration of each loop** = 600 s
- **number of calculation statements** = 50 **"result=(CONSTANT_VALUE/variable_value);"**-Instructions
  (executed twice per pin cycle)
- **frequencyA** as measured with a universal counter
- **frequencyB** as measured with a universal counter

The reference loop has the same structure as the measurement loop except that the measurement sequence is moved.

State	jPS-Value (mean)
TC65 module in IDLE mode / Not connected	~49000
CSD connection	~46000

Since only a small amount of Java code is executed in this test, it is easily optimized by the CLDC's HI just-in-time compiler. More complex applications might not reach that execution speed.

5.9.2 Pin I/O

The pin I/O test was designed to find out how fast a Java MIDlet can process URCs caused by Pin I/O and react to these URCs. The URCs are generated by feeding an input pin with an external frequency. As soon as the Java MIDlet is informed about the URC, it tries to regenerate the feeding frequency by toggling another output pin.

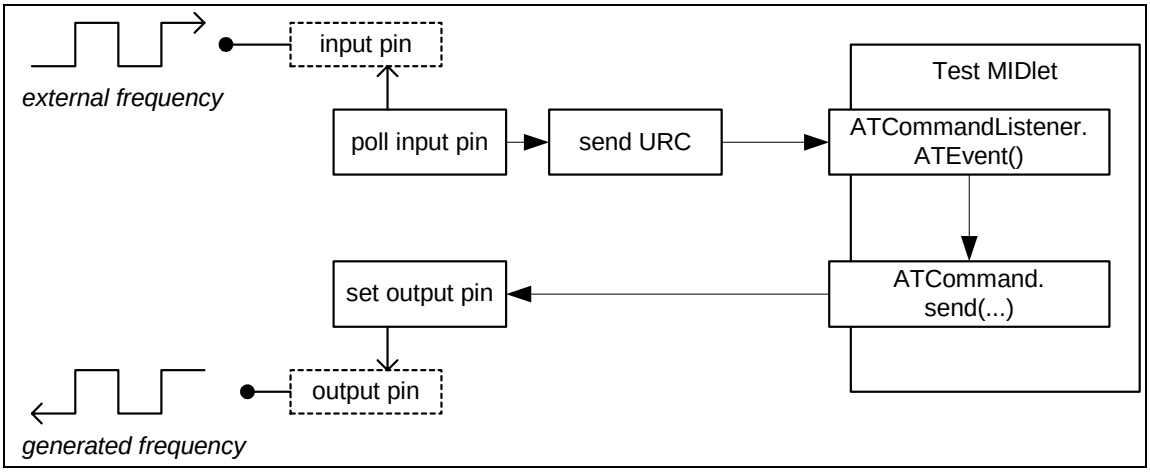


Figure 12: Test case for measuring Java MIDlet performance and handling pin-IO

The results of this test show that the delay from changing the state on the input pin to a state change on the output pin is at least around 50 ms, but that time strongly depends on the amount of garbage to collect and number of threads to be served by the virtual machine. Consequently, pin I/O is not suitable for generating or detecting frequencies.

5.9.3 Data Rates on RS-232 API

For details about the software platform and interfaces refer to Chapter 4, "Software Platform". This section summarises limitations and preconditions for performance when using the interface CommConnection from package com.siemens.mp.io (refer to [5]).

The data rate on RS232 depends on the size of the buffer used for reading from and writing to the serial interface. It is recommended that method *read (byte[ ] b)* be used for reading from the serial interface. The recommended buffer size is 2kbyte. To achieve error free data transmission the flow control on CommConnection must be switched on: <autorts> and <autocts>, the same for the connected device.

Different use cases are listed to give an idea of the attainable data rates. All applications for measurement use only one thread and no additional activities other than those described were carried out in parallel.

5.9.3.1 Plain Serial Interface

Scenario: A device is connected to ASC0 (refer to 4.2.4). The Java application must handle data input and output streams. A simple Java application (with only one thread) which loops incoming data directly to output, reaches data rates up to 180kbit/s. Test conditions: hardware flow control enabled (<autorts> and <autocts>), 8N1, and baud rate on ASC0 set to 230kbaud (-> theoretical maximum: 184kbit/s net data rate).

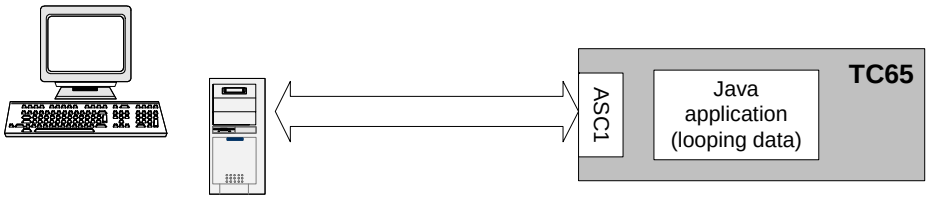


Figure 13: Scenario for testing data rates on ASC1

5.9.3.2 Voice Call in Parallel

Same scenario as in section 5.9.3.1, but with a voice call added. The application reflects incoming data directly to output and, additionally, handles an incoming voice call. The data rates are also up to 180kbit/s. Test conditions: same as in Sect. 5.9.3.1.

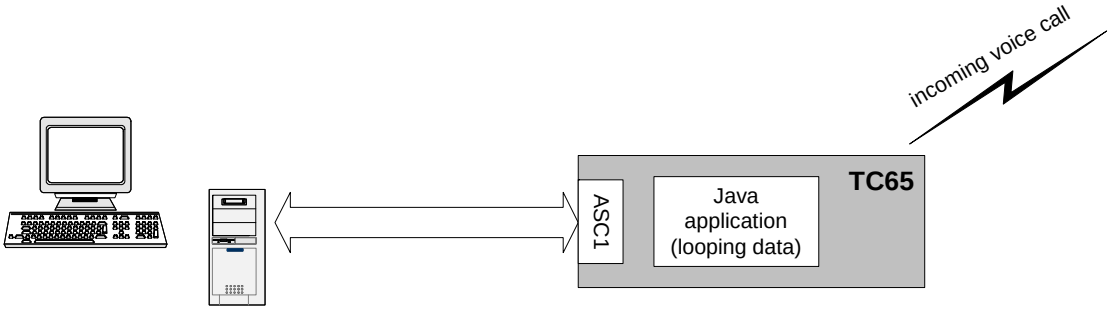


Figure 14: Scenario for testing data rates on ASC1 with a voice call in parallel

5.9.3.3 Scenarios with GPRS Connection

The biggest challenges to the module performance are setting up a GPRS connection, receiving data on javax.microedition.io interfaces and sending or receiving the data on the RS232 API with the help of a Java application.

5.9.3.3.1 Upload

Since the TC65 supports GPRS class 12, up to four timeslots for upload data are available. The Java application receives data over RS232 API and sends them over GPRS to a server.

Table 1: Download data rate with different number of timeslots, CS2

Upload data rate with x timeslots *Coding scheme 2* [kbit/s]											
1 timeslot	**theor. Value ***	**% from theor. Value**	**2 time-slots**	**theor. Value ***	**% from theor. Value**	**3 time-slots**	**theor. Value ***	**% from theor. Value**	**4 time-slots**	**theor. Value ***	**% from theor. Value**
9	12	75%	15	24	63%	20	36	55%	16	48	33%

* net transmission rates for LLC layer

Table 2: Download data rate with different number of timeslots, CS4

Upload data rate with x timeslots *Coding scheme 4* [kbit/s]											
1 timeslot	**theor. Value ***	**% from theor. value**	**2 time-slots**	**theor. Value ***	**% from theor. value**	**3 time-slots**	**theor. Value ***	**% from theor. value**	**4 time-slots**	**theor. Value ***	**% from theor. value**
13	20	65%	22	40	55%	20	60	33%	13	80	16%

* net transmission rates for LLC layer

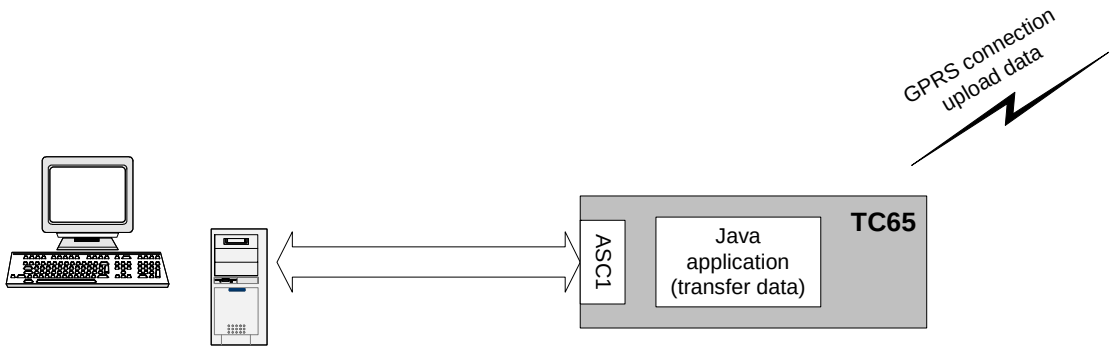


Figure 15: Scenario for testing data rates on ASC1 with GPRS data upload

5.9.3.3.2 Download

The data rate for downloading data over GPRS depends on the number of assigned timeslots and the coding schemes given by the net. Since TC65 supports GPRS class 12, the number of assigned timeslots can be up to 4. For the measurements, the Java application receives data from the server over GPRS and sends them over RS232 to an external device.

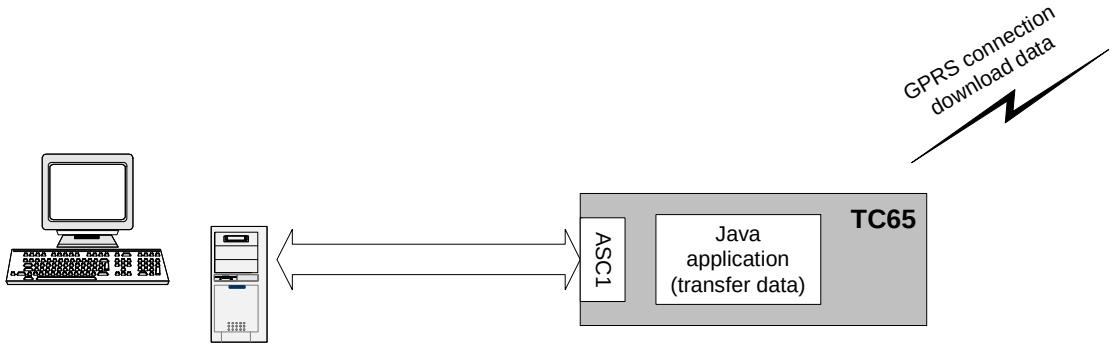


Figure 16: Scenario for testing data rates on ASC1 with GPRS data download

The tables below show the download data rates that can be achieved if hardware flow control is enabled on the CommConnection interface and the port speed is set to 230400.

Table 3: Download data rate with different number of timeslots, CS2

Download data rate with x timeslots *Coding scheme 2* [kbit/s]											
1 timeslot	**theor. Value ***	**% from theor. Value**	**2 time-slots**	**theor. Value ***	**% from theor. Value**	**3 time-slots**	**theor. Value ***	**% from theor. Value**	**4 time-slots**	**theor. Value ***	**% from theor. Value**
11	12	91%	21	24	87%	29	36	81%	35	48	73%

* net transmission rates for LLC layer

Table 4: Download data rate with different number of timeslots, CS4

Download data rate with x timeslots *Coding scheme 4* [kbit/s]											
1 timeslot	**theor. Value ***	**% from theor. value**	**2 time-slots**	**theor. Value ***	**% from theor. value**	**3 time-slots**	**theor. Value ***	**% from theor. value**	**4 time-slots**	**theor. Value ***	**% from theor. value**
17	20	85%	31	40	78%	35	60	58%	38	80	48%

* net transmission rates for LLC layer

6 MIDlets

The J2ME™ Mobile Information Device Profile (MIDP) provides a targeted Java API for writing wireless applications. The MIDP runs on top of the Connected Limited Device Configuration (CLDC), which in turn, runs on top of the J2ME™. MIDP applications are referred to as MIDlets. MIDlets are controlled by the mobile device implementation that supports the CLDC and MIDP. Since IMP-NG is a subset of MIDP 2.0, IMP includes MIDlets. The MIDlet code structure is very similar to applet code. There is no main method and MIDlets always extend from the MIDlet class. The MIDlet class in the MIDlet package provides methods to manage a MIDlet's life cycle.

6.1 MIDlet Documentation

MIDP and MIDlet documentation can be found at http://wireless.java.sun.com/midp/ and in the html document directory of the wtk,
…\Siemens\SMTK\TC65\wtk\doc\index.html

6.2 MIDlet Life Cycle

The MIDlet life cycle defines the protocol between a MIDlet and its environment through a simple well-defined state machine, a concise definition of the MIDlet's states and APIs to signal changes between the states. A MIDlet has three valid states:

- *Paused* – The MIDlet is initialised and is quiescent. It should not be holding or using any shared resources.
- *Active* – The MIDlet is functioning normally.
- *Destroyed* – The MIDlet has released all of its resources and terminated. This state is only entered once.

State changes are affected by the MIDlet interface, which consists of:
- *pauseApp()* – the MIDlet should release any temporary resources and become passive.
- *startApp()* – the MIDlet starts it's execution, needed resources can be acquire here or in the MIDlet constructor
- *destroyApp()* – the MIDlet should save any state and release all resources
- Note: destroyApp() is called when a MIDlet should terminate caused by device.
- *notifyDestroyed()* – the MIDlet notifies the application management software that it has cleaned up and is done
- Note: the only way to terminate a MIDlet is to call notifyDestroyed(), but destroyApp() is not automatically called by notifyDestroyed().
- *notifyPaused()* – the MIDlet notifies the application management software that it has paused

- *resumeRequest()* – the MIDlet asks application management software to be started again.

    - *getAppProperty()* – gets a named property from the MIDlet

Table 5: A typical sequence of MIDlet execution

Application Management Software	MIDlet
The application management software creates a new instance of a MIDlet.	The default (no argument) constructor for the MIDlet is called; it is in the Paused state.
The application management software has decided that it is an appropriate time for the MIDlet to run, so it calls the *MIDlet.startApp* method for it to enter the Active state.	The MIDlet acquires any resources it needs and begins to perform its service.
The application management software no longer needs the application be active, so it signals it to stop performing its service by calling the *MIDlet.pauseApp* method.	The MIDlet stops performing its service and might choose to release some resources it currently holds.
The application management software has determined that the MIDlet is no longer needed, or perhaps needs to make room for a higher priority application in memory, so it signals the MIDlet that it is a candidate to be destroyed by calling the *MIDlet.destroyApp* method.	If it has been designed to do so, the MIDlet saves state or user preferences and performs clean up.

6.3 Hello World MIDlet

Here is a sample HelloWorld program.

```
/**
 * HelloWorld.java
 */

package example.helloworld;
import javax.microedition.midlet.*;
import java.io.*;

public class HelloWorld extends MIDlet {

  /**
   * HelloWorld - default constructor
   */
  public HelloWorld() {
    System.out.println("HelloWorld: Constructor");
  }

  /**
   * startApp()
   */
  public void startApp() throws MIDletStateChangeException {
    System.out.println("HelloWorld: startApp");
    System.out.println("\nHello World!\n");
    destroyApp();
  }

  /**
   * pauseApp()
   */
  public void pauseApp() {
    System.out.println("HelloWorld: pauseApp()");
  }

  /**
   * destroyApp()
   */
  public void destroyApp(boolean cond) {
    System.out.println("HelloWorld: destroyApp(" + cond + ")");
    notifyDestroyed();
  }
}
```

7 File Transfer to Module

7.1 Module Exchange Suite

The Module Exchange Suite allows you to view the Flash file system on the module as a directory from Windows Explorer. Make sure that the module is turned on and that one of the module's serial interfaces (ASC0, ASC1 or USB) is connected to the COM port that the Module Exchange Suite is configured to. The configured COM port can be checked or changed under Properties of the Module directory. Please note that the Module Exchange Suite can be used only if the module is in normal mode. While running the module with the Module Exchange Suite, subdirectories and files can be added to the flash file system of module. Keep in mind that a maximum of 200 flash objects (files and subdirectories) per directory in the flash file system of the module is recommended.

7.1.1 Windows Based

The directory is called "Module" and can be found at the top level of workspace "MyComputer". To transfer a file to the module, simply copy the file from the source directory to the target directory in the "Module -> Module Disk (A:)".

7.1.2 Command Line Based

A suite of command line tools is available for accessing the module's Flash file system. They are installed in the Windows System directory so that the tools are available from any directory. The module's file system is accessed with *mod:*. The tools included in this suite are MESdel, MEScopy, MESxcopy, MESdir, MESmkdir, MESrmdir, MESport, MESclose and MESformat. Entering one of these commands without arguments will describe the command's usage. The tools mimic the standard directory and file commands. A path inside the module's file system is identified by using "mod:" followed by the module disk which is always "A:" (e.g. "MESdir mod:a:" lists the contents of the module's root directory).

7.2 Over the Air Provisioning

See Chapter 8 for OTA provisioning.

7.3 Security Issues

The developer should be aware of the following security issues. Security aspects in general are discussed in chapter 11.

7.3.1 Module Exchange Suite

The serial interface should be mechanically protected.

The copy protection rules for Java applications prevent opening, reading, copying, moving or renaming of JAR files. It is not recommended that the name of a Java application (for example <name>.jar) be used for a directory, since the copy protection will refuse access to open, copy or rename such directories.

7.3.2 OTAP

- A password should be used to update with OTA (SMS Authentication)
- Parameters should be set to fixed values (***at^sjotap***) whenever possible so that they cannot be changed over the air.
- The http server should be secure. (e.g. Access control via basic authentication)

8 Over The Air Provisioning (OTAP)

8.1 Introduction to OTAP

OTA (Over The Air) Provisioning of Java Applications is a common practice in the Java world. OTAP describes mechanisms to install, update and delete Java applications over the air. The TC65 product implements the Over The Air Application Provisioning as specified in the IMP-NG standard (JSR228).

The OTAP mechanism described in this document does not require any physical user interaction with the device; it can be fully controlled over the air interface. Therefore it is suitable for Java devices that are designed not to require any manual interaction such as vending machines or electricity meters.

8.2 OTAP Overview

To use OTAP, the developer needs, apart from the device fitted with the TC65 module, an http server, which is accessible over a TCP/IP connection either over GPRS or CSD, and an SMS sender, which can send Class1, PID $7d short messages. This is the PID reserved for a module's data download.

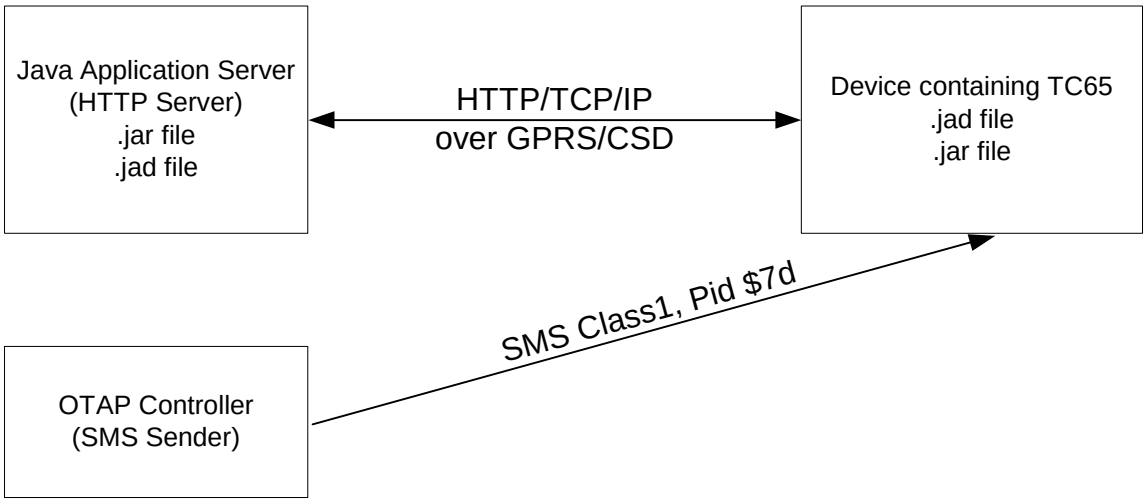


Figure 17: OTAP Overview

The Java Application Server (http Server) contains the .jar and the .jad file to be loaded on the device. Access to these files can be protected by http basic authentication.

The OTAP Controller (SMS Sender) controls the OTAP operations. It sends SMs, with or without additional parameters, to the devices that are to be operated. These devices then try to contact the http server and download new application data from it. The OTAP Controller will not get any response about the result of the operation. Optionally the server might get a result response over http.

There are two types of OTAP operations:

- Install/Update: A new JAR and JAD file are downloaded and installed.
- Delete: A complete application (.jar, .jad, all application data and its directory) is deleted.

8.3 OTAP Parameters

There is a set of parameters that control the OTAP procedures. These parameters can either be set by AT command (**at^sjotap**, refer to AT Command Set [7]) during the production of the device, or by SM (see Section 8.4) during operation of the device in the field. None of the parameters, which are set by AT command, can be overwritten by SM.

- JAD File URL: the location of the JAD file is used to install or update procedures. The JAD file needs to be located on the net (e.g. http://someserver.net/somefile.jad or http://192.168.1.2/somefile.jad).
- Application Directory: this is the directory where a new application (JAD and JAR file) is installed. The delete operation deletes this directory completely. When entering the application directory with **at^sjotap** or a short message ensure that the path name is not terminated with a slash. For example, type "a:" or "a:/otap" rather than "a:/" or "a:/otap/". See examples provided in Chapter 8.4.
- http User: a username used for authentication with the http server.
- http Password: a password used for authentication with the http server.
- Bearer: the network bearer used to open the HTTP/TCP/IP connection, either GPRS or CSD.
- APN or Number: depending on the selected network bearer this is either an access point name for GPRS or a telephone number for CSD.
- Net User: a username used for authentication with the network.
- Net Password: a password used for authentication with the network.
- DNS: a Domain Name Server's IP address used to query hostnames.
- NotifyURL: the URL to which results are posted. This parameter is only used when the MIDlet-Install-Notify attribute or MIDlet-Delete-Notify attribute is not present in the descriptor.

There is one additional parameter that can only be set by AT command:
- SM Password: it is used to authenticate incoming OTAP SMs. Setting this password gives an extra level of security.
  Note: If a password set by AT command, all SMs must include this password

Table 6: Parameters and keywords

Parameters	Max. Length AT	Keyword SM	Install/update	delete
JAD File URL	*100*	JADURL	mandatory	unused
Application Directory	*50*	APPDIR	mandatory	mandatory
HTTP User	*32*	HTTPUSER	optional	unused
HTTP Password	*32*	HTTPPWD	optional	unused
Bearer	--	BEARER	mandatory	optional/P
APN or Number	*65*	APNORNUM	mandatory for CSD	optional/P
Net User	*32*	NETUSER	optional	optional/P
Net Password	*32*	NETPWD	optional	optional/P
DNS	--	DNS	optional	optional/P
Notify URL	*100*	NOTIFYURL	optional	optional/P
SM Password	*32*	PWD	optional	optional

The length of the string parameters in the AT command is limited (see Table 6), the length in the SM is only limited by the maximum SM length.

The minimum set of required parameters depends on the intended operation (see Table 6). “optional/P” indicates that this parameter is only necessary when a POST result is desired.

8.4 Short Message Format

An OTAP control SM must use a Submit PDU with Class1, PID $7d and 8 bit encoding. As a fallback for unusual network infrastructures the SM can also be of Class0 and/or PID $00. The content of the SM consists of a set of keywords and parameter values all encoded in ASCII format. These parameters can be distributed over several SMs. There is one single keyword to start the OTAP procedure. For parameters that are repeated in several SMs only the last value sent is valid. For example, an SM could look like this:

Install operation:

First SM:
```
OTAP_IMPNG
PWD:secret
JADURL:http://www.greatcompany.com/coolapps/mega.jad
APPDIR:a:/work/appdir
HTTPUSER:user
HTTPPWD:anothersecret
```

Second SM:
```
OTAP_IMPNG
PWD:secret
BEARER:gprs
APNORNUM:access.to-thenet.net
NETUSER:nobody
NETPWD:nothing
DNS:192.168.1.2
START:install
```

Delete operation:

```
OTAP_IMPNG
PWD:secret
APPDIR:a:/work/appdir
START:delete
```

The first line is required: it is used to identify an OTAP SM. All other lines are optional and their order is insignificant, each line is terminated with an LF: '\n' including the last one. The keywords, in capital letters, are case sensitive. A colon separates the keywords from their values.

The values of APPDIR, BEARER and START are used internally and must be lower case. The password (PWD) is case sensitive. The case sensitivity of the other parameter values depends on the server application or the network. It is likely that not all parameters can be sent in one SM. They can be distributed over several SMs. Every SM needs to contain the identifying first line (OTAP_IMPNG) and the PWD parameter if a mandatory password has been enabled. OTAP is started when the keyword START, possibly with a parameter, is contained in the SM and the parameter set is valid for the requested operation. It always ends with a reboot, either when the operation is completed, an error occurred, or the safety timer expired. This also means that all parameters previously set by SM are gone.

Apart from the first and the last line in this example, these are the parameters described in the previous section. Possible parameters for the START keyword are: “install”, “delete” or nothing. In the last case, an install operation is done by default.

The network does not guarantee the order of SMs. So when using multiple SMs to start an OTAP operation their order on the receiving side might be different from the order in which they were sent. This could lead to trouble because the OTAP operation might start before all parameters are received. If you discover such problems, try waiting a few seconds between each SM.

8.5 Java File Format

In general, all Java files have to comply with the IMP-NG and TC65 specifications. There are certain components of the JAD file that the developer must pay attention to when using OTAP:

- MIDlet-Jar-URL: make sure that this parameter points to a location on the network where your latest JAR files will be located, e.g. http://192.168.1.3/datafiles/mytest.jar, not in the filesystem like file://a:/java/mytest/mytest.jar. Otherwise this JAD file is useless for OTAP.
- MIDlet-Install-Notify: this is an optional entry specifying a URL to which the result of an update/install operation is posted. That is the only way to get any feedback about the outcome of an install/update operation. The format of the posted URL complies with the IMP-NG OTA Provisioning specification. In contrast to the jar and jad file this URL must not be protected by basic authentication.
- MIDlet-Delete-Notify: this is an optional entry specifying a URL to which the result of a delete operation is posted. That is the only way to get any feedback about the outcome of a delete operation. The format of the posted URL complies with the IMP-NG OTA Provisioning specification. In contrast to the jar and jad file this URL must not be protected by basic authentication.
- MIDlet-Name, MIDlet-Version, MIDlet-Vendor: are mandatory entries in the JAD and Manifest file. Both files must contain equal values, otherwise result 905 (see 8.7) is returned.
- MIDlet-Jar-Size must contain the correct size of the jar file, otherwise result 904 (see 8.7) is returned.

Example:
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Description: My very important test
MIDlet-1: MyTest, , example.mytest.MyTest
MIDlet-Jar-Size: 1442
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1

A suitable Manifest file for the JAD file above might look like:
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: TLR Inc.
MIDlet-1: MyTest, , example.mytest.MyTest
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1

8.6 Procedures

8.6.1 Install/Update

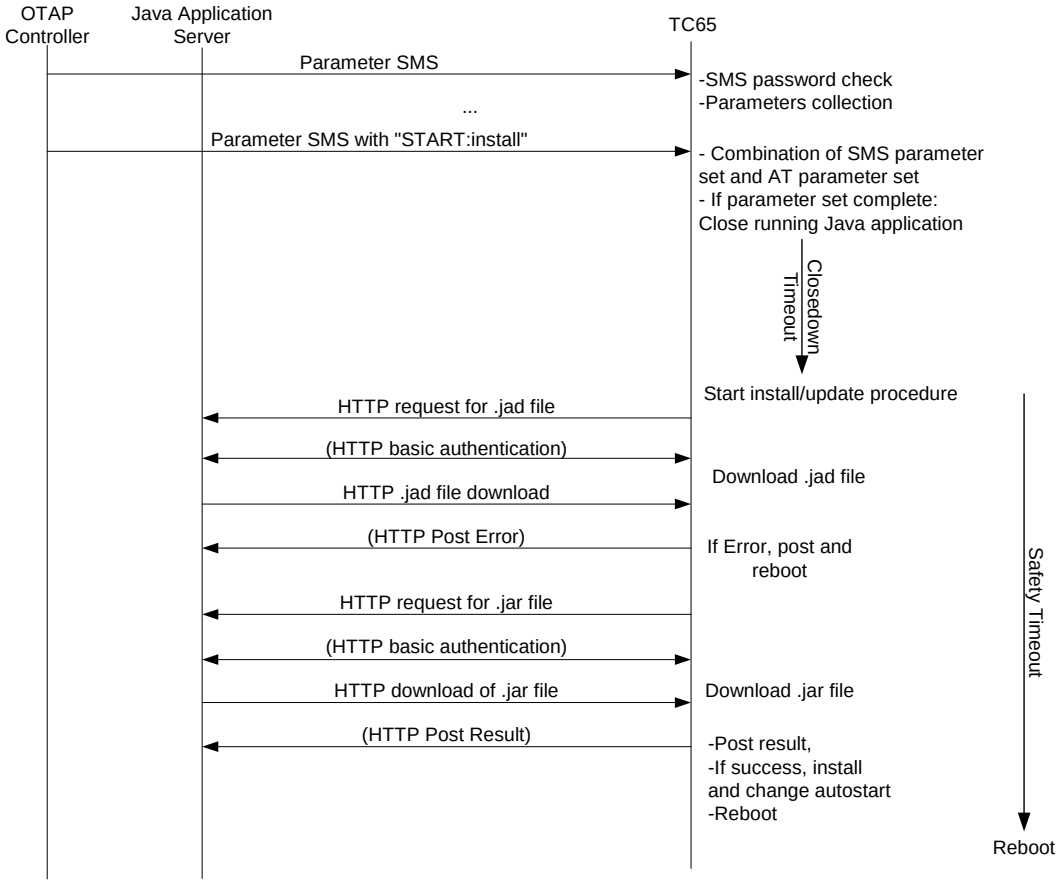


Figure 18: OTAP: Install/Update Information Flow

(The messages in brackets are optional)

When an SM with keyword START:install is received and there is a valid parameter set for the operation, the module always reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error during an update operation the old application is kept untouched, with one exception. If there is not enough space in the file system to keep the old and the new application at the same time, the old application is deleted before the download of the new one, therefore it is lost when an error occurs.
If install/update was successful, autostart is set to the new application.

8.6.2 Delete

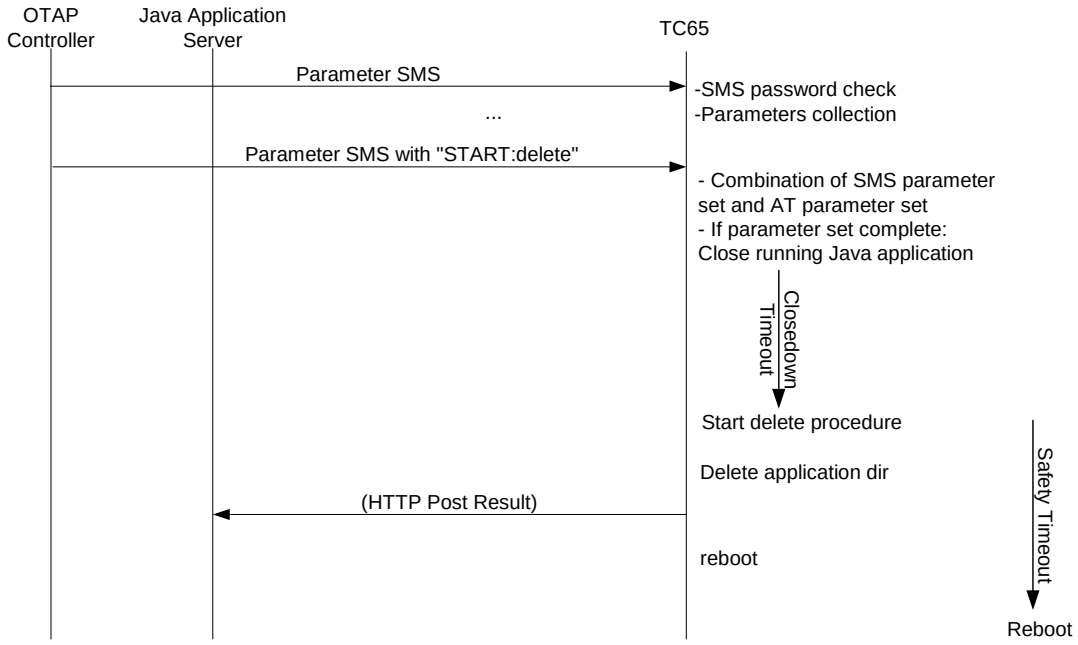


Figure 19: OTAP: Delete Information Flow

(The messages in brackets are optional)

When an SM with keyword START:delete is received and there is a valid parameter set for this operation, the module reboots either when the operation completed, an error occurred or the safety timer expired. If there is any error the application is kept untouched. Autostart is not changed.

8.7 Time Out Values and Result Codes

Timeouts:
- Closedown Timeout: 10 seconds
- Safety Timeout: 10 minutes

Result Codes: Supported status codes in body of the http POST request:
- 900 Success
- 901 Insufficient memory in filesystem
- 902 -not supported-
- 903 -not supported-
- 904 JAR size mismatch, given size in JAD file does not match real size of jar file
- 905 Attribute mismatch, one of the mandatory attributes MIDlet-name, MIDlet-version, MIDlet-Vendor in the JAD file does not match those given in the JAR manifest
- 906 invalid descriptor, something is wrong with the format of the .jad file
- 907 invalid JAR, the JAR file was not available under MIDlet-Jar-URL, files could not be extracted from JAR archive, or something else is wrong with the format of the file.
- 908 incompatible configuration or profile
- 909 application authentication failure, signature did not match certificate
- 910 application authorization failure, tried to replace signed with unsigned version
- 911 -not supported-
- 912 Delete Notification

All HTTP packets (GET, POST) sent by the module contain the IMEI number in the User-Agent field, e.g.

```
User-Agent: TC65/000012345678903 Profile/IMP-NG Configuration/CLDC-1.1
```

This eases device identification at the HTTP server.

8.8 Tips and Tricks for OTAP

- For security reasons it is recommended that an SMS password be used. Otherwise the “delete” operation can remove entire directories without any authentication.
- For extra security, set up a private CSD/PPP Server and set its phone number as a fixed parameter. This way, applications can only be downloaded from one specific server.
- As a side effect, OTAP can be used to simply reboot the module. Simply start an OTAP procedure with a parameter set which will not really do anything, such as a delete operation on a nonexistent directory.
- If you do not want to start OTAP by SMS let your Java application do it by issuing the **at^sjotap** command. This triggers an install/update operation as described in chapter 8.6.1 but without the SMS part.
  Note: If a malfunctioning Java application is loaded the SM method will still be needed for another update.
- The OTAP procedure cannot be tested in the debug environment
- Be aware that the module needs to be logged into the network to do OTAP. That means that either the Java application must enter the PIN, the PIN needs to be disabled or Autopin (see AT Command Set [3]) needs be used.
- The OTAP procedure might fail due to call collision, e.g. an incoming call when OTAP tries to start a CSD connection.

8.9 OTAP Tracer

For easy debugging of the OTAP scenario, the OTAP procedure can be traced over the serial interface. The trace output shows details of the OTAP procedure and the used parameters. To enable the OTAP trace output use the at command **at^scfg**, e.g.

AT^SCFG=Trace/Syslog/OTAP,1

The serial interface on which you issue this command is then exclusively used for the OTAP tracer. All other functionality which is normally present (AT commands or CommConnection and System.out in Java) is not available when the tracer is on.

This feature is intended to be used during development phase and not in deployed devices.

8.10 Security

Java Security as described in chapter 11 also has consequences for OTAP. If the module is in secured mode the MIDlet signature is also relevant to the OTAP procedure. This means:

- if the application is a unsigned version of a installed signed version of the same application then status code 910 is returned
- if the applications signature does not match the module's certificate then status code 909 is returned

8.11 How To

This chapter is a step-by-step guide for using OTAP.

1. Do you need OTAP? Is there any chance that it might be necessary to update the Java application, install a new one or delete it? It could be that device is in the field and you cannot or do not want to update it over the serial line. If the answer is “yes” then read through the following steps, if the answer is “no” then consider simply setting the OTAP SMS password to protect your system. Then you are finished with OTAP.
2. Take a look at the parameters (chapter 8.3), which control OTAP. You need to decide which of them you want to allow to be changed over the air (by SMS) and which you do not. This is mainly a question of security and what can fit into a short message. Then set the “unchangeable” parameters with the AT command (*at^sjotap*).
3. Prepare the http server. The server must be accessible from your device over TCP/IP. That means there is a route from your device over the air interface to the http server and back. When in doubt, write a small Java application using the httpConnection Interface to test it.
4. Prepare the JAR and JAD files which are to be loaded over the air. Make sure that these files conform to the requirements listed in chapter 8.5 and that they represent a valid application which can be started by *at^sjra***.**
5. Put the files (JAR and JAD) on the http Server. The files can either be publicly available or protected through basic authentication. When in doubt try to download the files from the server by using a common web browser on a PC, which can reach your http server over TCP/IP.
6. Prepare the SM sender. The sender must be able to send SMs, which conform to chapter 8.4, to your device. When in doubt try to send “normal” SMs to your device which can than be read out from the AT command interface.
7. Test with a local device. Send a suitable short message to your device, which completes the necessary parameter, sets and starts the operation. The operation is finished when the device reboots. You can now check the content of the file system and if the correct jar and jad files were loaded into the correct location.
8. Analyze errors. If the above test failed, looking at your device’s behavior and your http servers access log can give you a hint as to what went wrong:
   - If the device did not terminate the running Java application and did not reboot, not even after the safety timeout, either your SM was not understood (probably in the wrong format) or it did not properly authenticate (probably used the wrong password) or your parameter set is incomplete for the requested operation.
   - If the device terminated the running Java application, but did not access your http server, and rebooted after the safety timeout, there were most likely some problems when opening the network connection. Check your network parameters.
   - If the device downloaded the jad and possibly even the jar file but then rebooted without saving them in the file system, most likely one of the errors outlined in chapter 8.5 occurred. These are also the only errors which will return a response. They are posted to the http server if the jad file contains the required URL.
9. Start update of remote devices. If you were able to successfully update your local device, which is hopefully a mirror of all your remote devices, you can start the update of all other devices.

9 Compile and Run a Program without a Java IDE

This chapter explains how to compile and run a Java application without a Java IDE.

9.1 Build Results

A JAR file must be created by compiling an SMTK project. A JAR file will contain the class files and auxiliary resources associated with an application. A JAD file contains information (file name, size, version, etc.) on the actual content of the associated JAR file. It must be written by the user. The JAR file has the ".jar" extension and the JAD file has the ".jad" extension. A JAD file is always required no matter whether the module is provisioned with the Module Exchange Suite, as described in Section 7.1, or with OTA provisioning. OTA provisioning is described in Chapter 8.

In addition to class and resource files, a JAR file contains a manifest file, which describes the contents of the JAR. The manifest has the name manifest.mf and is automatically stored in the JAR file itself. An IMP manifest file for:

```
package example.mytest;
public class MyTest extends MIDlet
```

includes at least:

```
Manifest-Version: 1.0
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Siemens
MIDlet-1: MyTest, example.mytest.MyTest
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

A JAD file must be written by the developer and must include at least:

```
MIDlet-Name: MyTest
MIDlet-Version: 1.0.1
MIDlet-Vendor: Siemens
MIDlet-1: MyTest, example.mytest.MyTest
MIDlet-Jar-URL: http://192.168.1.3/datafiles/MyTest.jar
MIDlet-Jar-Size: 1408
MicroEdition-Profile: IMP-NG
MicroEdition-Configuration: CLDC-1.1
```

A detailed description of these attributes and others can be found in the Java/MIDlet documentation http://java.sun.com/j2me/docs/alt-html/WTK104_UG/Ap_Attributes.html

9.2 Compile

- Launch a *Command Prompt*. This can be done from the *Programs* menu or by typing "cmd" at the *Run…* prompt in the *Start* menu.
- Change to the directory where the code to be compiled is kept.
- Compile the program with the SDK. Examples of build batch files can be found in each sample program found in the examples directory, *\Siemens\SMTK\TC65\wtk\src\example.*
- If the compile was successful the program can be run from the command line. Examples of run batch files can be found in the examples directories listed above as well.

The batch files for compiling and running the samples refer to master batch files in the *…\Siemens\SMTK\TC65\wtk\bin* directory and use the system environment variables IMPNG_JDK_DIR and IMPNG_DIR. IMPNG_JDK_DIR points to the root directory of the installed JDK and IMPNG_DIR points to the root directory of the Siemens-SMTK-TC65-IMPNG installation. The installation process sets these environment variables. A modification is usually not necessary. They may be modified as requested (e.g. when switching to a different JDK) via the advanced system properties.

9.3 Run on the Module with Manual Start

- Compile the application at the prompt as discussed in Section 9.2 or in an IDE.
- Transfer the .jar and .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. Chapter 7 explains how to download your application to the module.
- Start a terminal program and connect to ASC0.
- The command *at^sjra* is used to start the application and is sent to the module via your terminal program. Either the application can be started by .jar or by .jad file.

Example:
In your terminal program enter: *at^sjra=a:/java/jam/example/helloworld/helloworld.jar*
If you prefer to start with .jad file: *at^sjra=a:/java/jam/example/helloworld/helloworld.jad*

The Flash file system on the module is referenced by "a:".
Depending on which file you specify the java application manager tries to find the corresponding file in the same directory. This search is not done by name, but by comparing the contained attributes. The first file which contains the same values for MIDlet-Name, MIDlet-Version and MIDlet-Vendor is used.

9.4 Run on the Module with Autostart

- Compile the application at the prompt as discussed in Section 9.2 or in an SMTK integrated IDE.
- Transfer the .jar and .jad file from the development platform to the desired directory on the module using the Module Exchange Suite or OTA provisioning. See Chapter 7.

9.4.1 Switch on Autostart

- There is an AT command, *at^scfg*, for configuring the autostart functionality. Please refer to the AT Command Set [3].
- Restart the module.

9.4.2 Switch off Autostart

There are two methods for switching off the autostart feature:
- the *at^scfg* AT command, or
- the “autostart_off.exe” tool (included in the Installation CD software under wtk/bin)

To disable the automatic start of a user application in a module these steps must be carried out:
1. Connect the module to the PC
2. Make sure, that the module is switched off
3. Start the Autostart_Off program
4. Select the COM-Port
5. Press the “Autostart Off” button

10 Debug Environment

Please note that this section is not intended as a tutorial in debugging or how to use Sun Java Studio, Borland JBuilder or Eclipse. Documents for these IDEs can be found on their respective homepages. Once the proper emulator has been selected (as described in the relevant IDE sections below), your Java application can be built, debugged and executed.

10.1 Data Flow of a Java Application in the Debug Environment

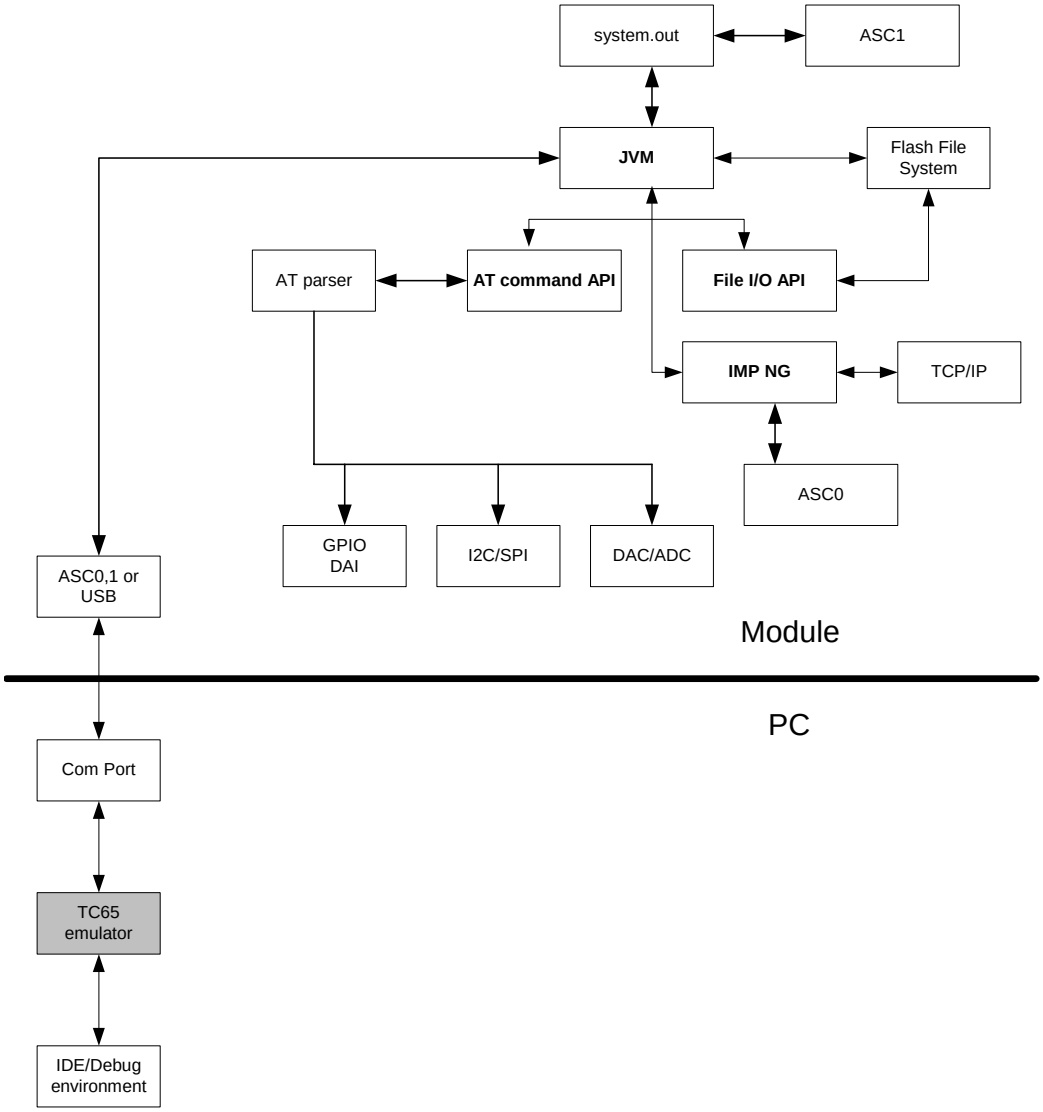


Figure 20: Data flow of a Java application in the debug environment

In the debug environment the module is connected to a PC via a serial interface. This can be a USB or an RS232 line. The application can then be edited, built, debugged or run within an IDE on the PC. When running or debugging the MIDlet under IDE control it is executed on the module (on-device execution) not on the PC. This can be either debugging mode, where the midlet execution can still be controlled from the IDE (on-device debugging) or normal mode, where the midlet is copied to the module and started normally. This ensures that all interfaces behave the same whether debugging mode is used or not.

10.2 Emulator

The TC65 emulator is part of the SMTK and is used as the controlling entity for on-device debugging. Before it can be used it must be configured:

The emulator accesses the module with AT commands. In order to do this, it must know the COM port and bit rate to use. The values are configured in the file wtk/bin/WM_Debug_config.ini. Set the “Port” and “Baudrate” parameters to the desired values.

Debugging information between the Debugger (IDE) and the JVM is transferred over an IP connection. In order to establish this IP connection between the PC and the module the emulator needs a special Dial-Up-Network (DUN) connection to be configured:

- ISP name: “IP connection for remote debugging”
- Modem: either “Standard 19200 bps Modem” or “TC65 14400 bps Modem”
- Phone number: *88#
- Username and password: any will do, it is recommended that the username and password be saved for re-usage.
- Disable the “Redial if line dropped” option.
- Enable “Connect automatically”

Make sure that the DUN uses the same COM port and bit rate as the emulator. You can use any of the three serial interfaces (ASC0, ASC1, USB) to connect with module, but you will lose the functionality which is normally present on the interface. Because of this loss and because of its speed it is recommended that the USB interface be used.

If necessary, the IP addresses used for the debug connection can also be changed. This is done in the file wtk/bin/WM_Debug_config.ini. See the documentation of **at^scfg** with parameter userware/debuginterface for details. Please keep in mind, that the IP address range 10.x.x.x is not supported for configuration of debugging!

During installation of TC65 SMTK some new programs are installed for handling the debugging session in conjunction with the IDE. The installation routine of the TC65 SMTK doesn't change any configuration of an existing firewall on your PC.
In the case, that a firewall is installed on your PC and the local configured and used IP connection (Dial-Up-Network connection for debugging) is blocked or disturbed by this firewall, please configure the firewall or the Dial-Up-Network connection manually to accept the new installed programs and the port or to use another port or contact your local PC administrator for help.

10.3 Java IDE

The SMTK is integrated into your Java IDE during installation. Please note that the IDE integration is intended to create MIDlets suitable for TC65 module and for debugging using the emulator. JAR files used in the module must be configured according to the batch file examples given. If the SMTK install succeeded, you can easily switch between the Siemens environment and Standard-JDK environment, the special libraries/APIs and emulators are available, and AT commands can be sent to the module. Regular function of the IDE for non-Siemens projects is unchanged.

Using the debugger please keep in mind that the MIDlet-URL, included in the Jad file, has to indicate and store the location where the TC65 emulator will find the corresponding Jar file. Generally the location will only be the file name for the Jar file.

If you are using Eclipse IDE the location of the Jar file is “deployed\<filename>.jar”. Please check this path name inside the Jad file before starting a debugging session with Eclipse IDE and change it manually, if the “deployed” subdirectory is missing. Please keep in mind, that the subdirectory “deployed” is used as a default setting in the Eclipse IDE and can be changed by the user within the Eclipse menu.

While using "on device debugging" the TC65 module is restarted after the end of each debugging session. This is independent of the used IDE (Eclipse 3.0.1, Eclipse 3.0.2, Sun Java Mobility Studio 6 2004Q3, JBuilder X, JBuilder 2005).

Please keep in mind, that it is not possible to use obfuscated files for a debugging session.

10.3.1 Sun Java Studio Mobility 6 2004Q3

This section indicates the changes to your IDE you will see after integrating the SMTK and describes how to exploit these features to build and debug your applications.

In the *Runtime Explorer*, Figure 21, the installed emulators can be seen under the *Device Emulator Registry*. The *Default Emulator* is set to the Siemens Emulator.

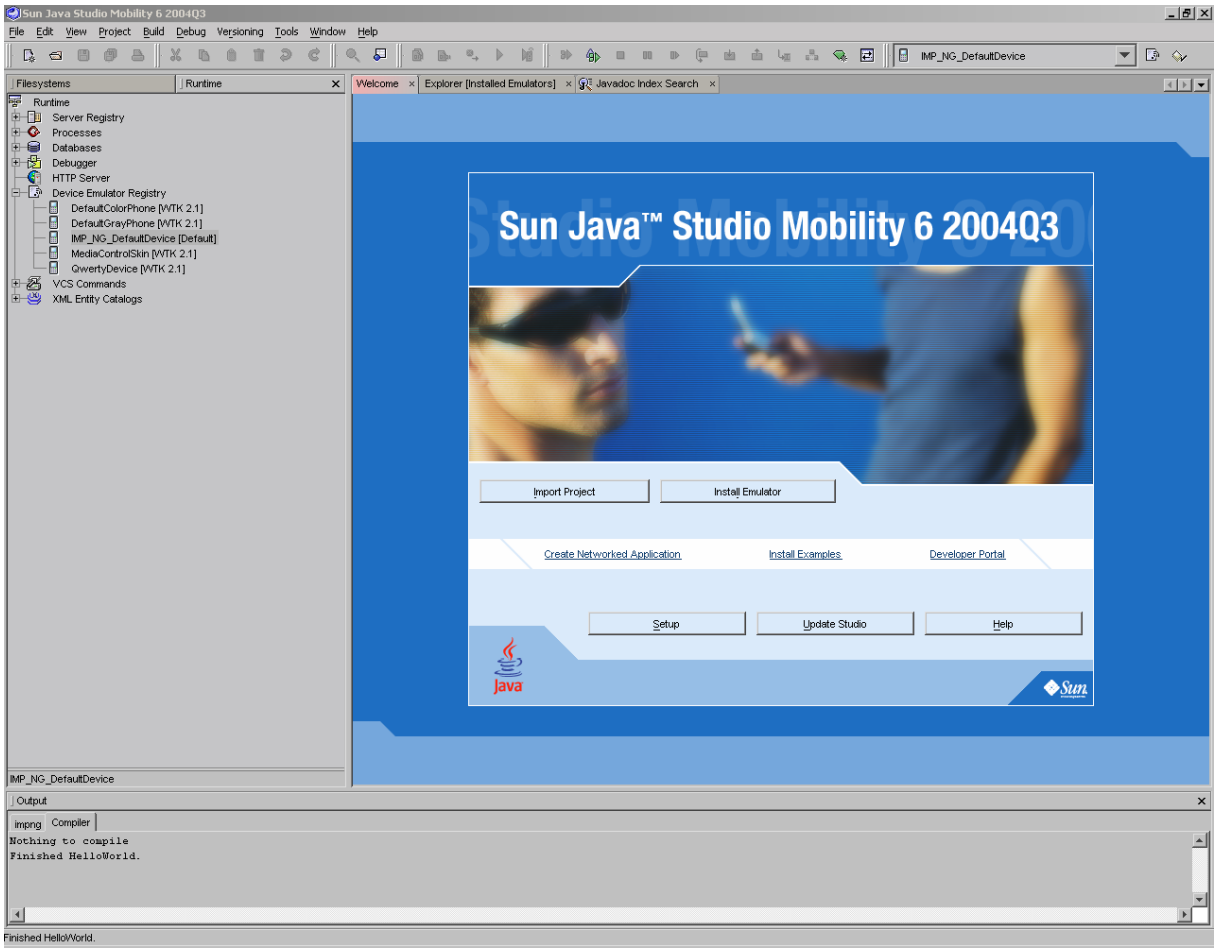


Figure 21: Sun Java Studio Mobility 6 - The installed emulators

10.3.1.1 Switching emulators

You can easily switch to the SMTK emulator by using the combo box in the main menu bar and choosing *IMP_NG_DefaultDevice*, see Figure 22. Any projects built and run when the Siemens emulator is selected will be compiled and run with the Siemens emulator. Directly after executing the IDE integration with the SMTK setup program the IMP_NG_DefaultDevice is activated by default.

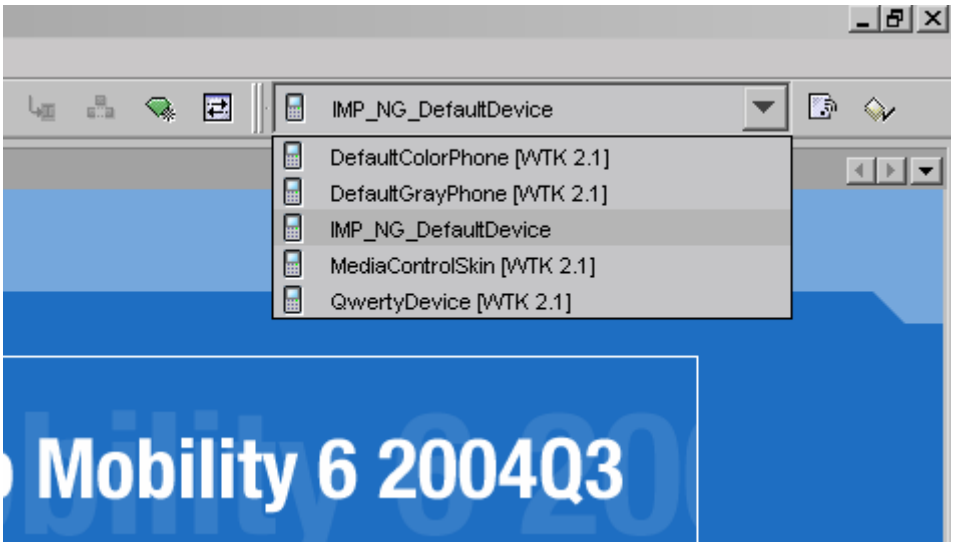


Figure 22: Sun Java Studio Mobility 6 - Switching Emulators

10.3.1.2 Projects

After integration, there is a new project in the *Project Manager*. This project contains the example and the additional libraries. The *Project Manager* is accessed through the *Project* menu. Directly after executing the IDE integration with the SMTK setup program the “Siemens TC65” project is opened by default.

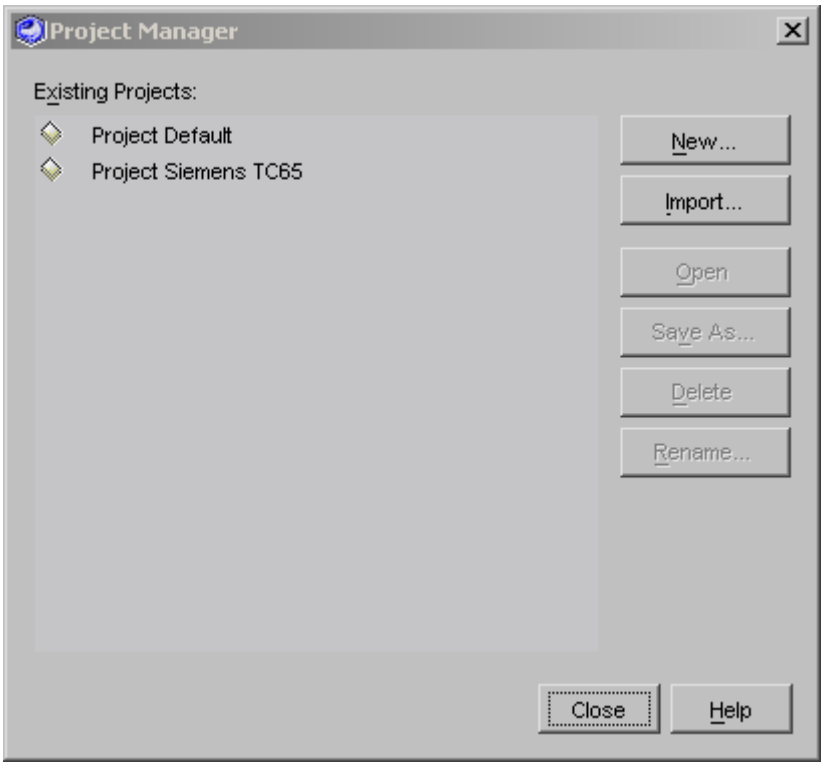


Figure 23: Sun Java Studio Mobility 6 - Project Manager

10.3.1.3 Templates

Templates for a Siemens MIDlet can be found in the file explorer and under File->New. The MIDlet template provides the skeleton of a MIDlet application.

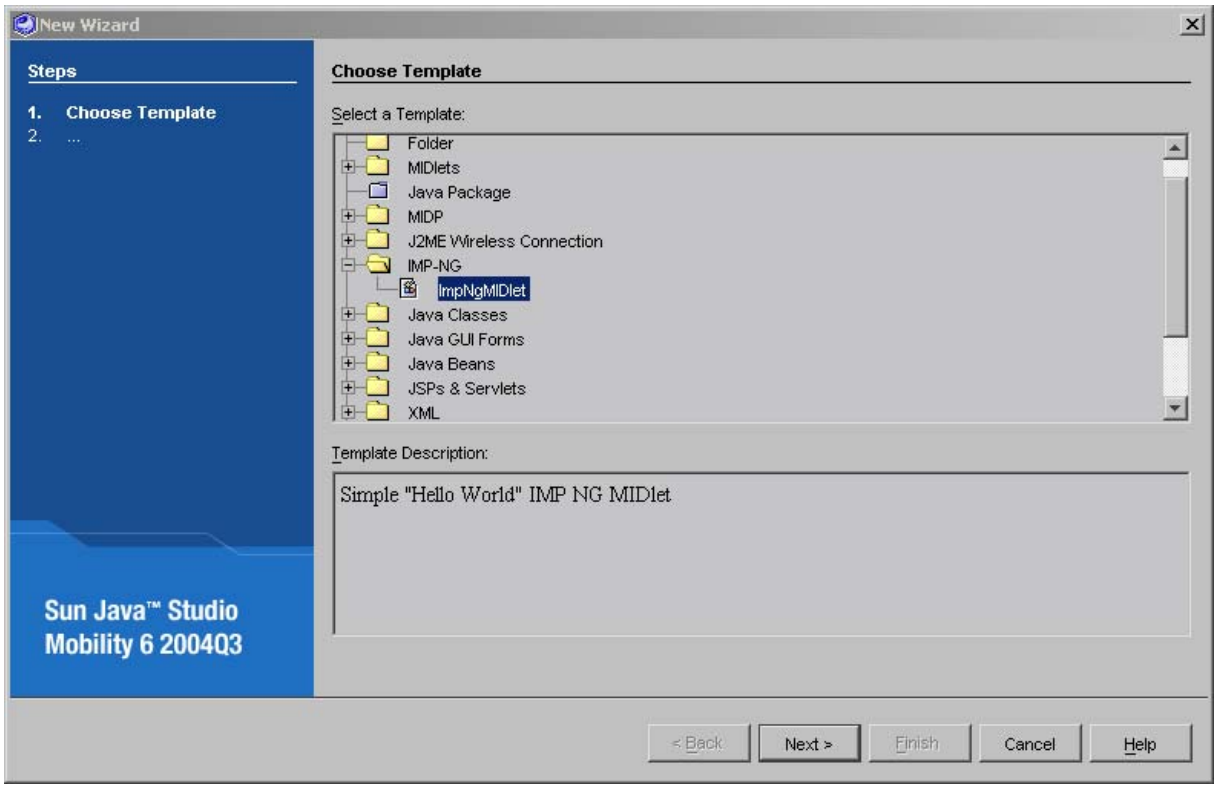


Figure 24: Sun Java Studio Mobility 6 - Selecting a template

10.3.1.4 Examples

There are sample MIDlets in the …\Siemens\SMTK\TC65\SUNStudioSamples directory.

10.3.1.5 Compile and run

Ensure that the proper emulator will be used and compile the project as normal. Any output will be shown in the output window in the IDE. The html help files of the SMTK can be accessed directly by pressing Alt+F1 or Shift+F1.

10.3.2 Borland JBuilder X

If you want to use JBuilderX and it is not installed, first install JBuilderX and follow the installation wizard instructions. Run the TC65 SMTK the installation program in maintenance mode. The SMTK will find the installed JBuilderX IDE. Select JBuilderX to integrate the SMTK into JBuilderX.

After integration of TC65 SMTK into JBuilderX you can examine the integration by opening the menu *Tools -> Configure JDKs…* (see Figure below)

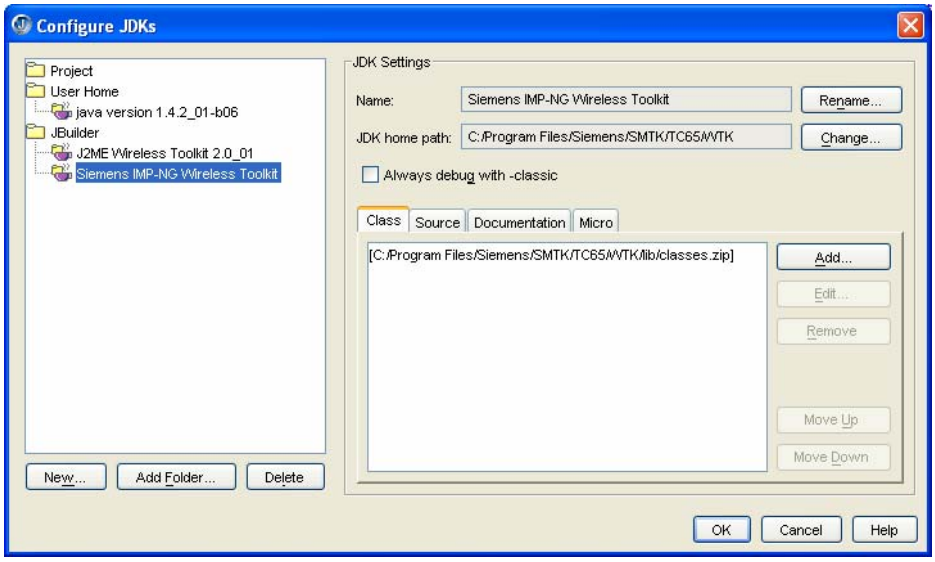


Figure 25: JBuilderX – JDK settings

The libraries included with the TC65 SMTK can be examined by opening the menu *Tools → Configure Libraries…* (see Figure below)

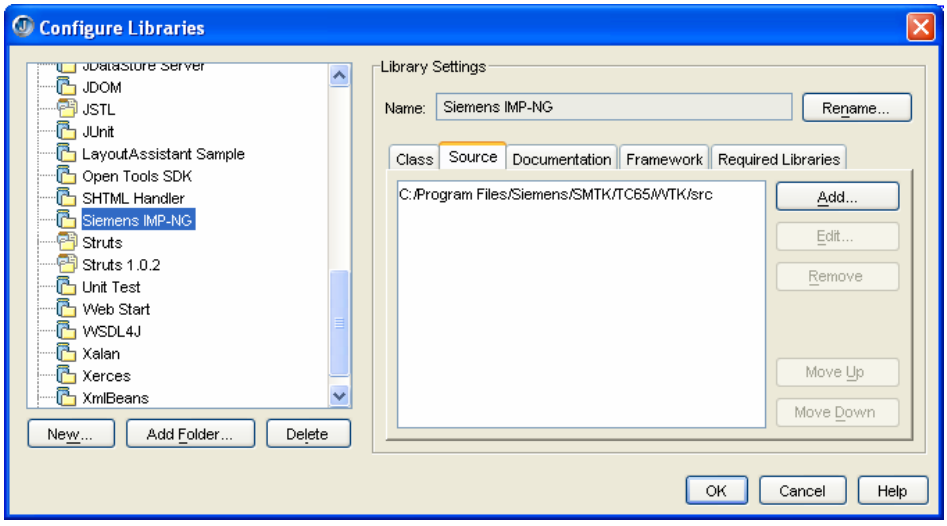


Figure 26: JBuilderX – Siemens Library

10.3.2.1 Examples

There are sample projects provided with the TC65 SMTK. These projects can be found in the JBuilderSamples directory of the TC65 SMTK installation directory. This directory is accessed by opening a project using the menu *File → Open Project…* (see Figure below)

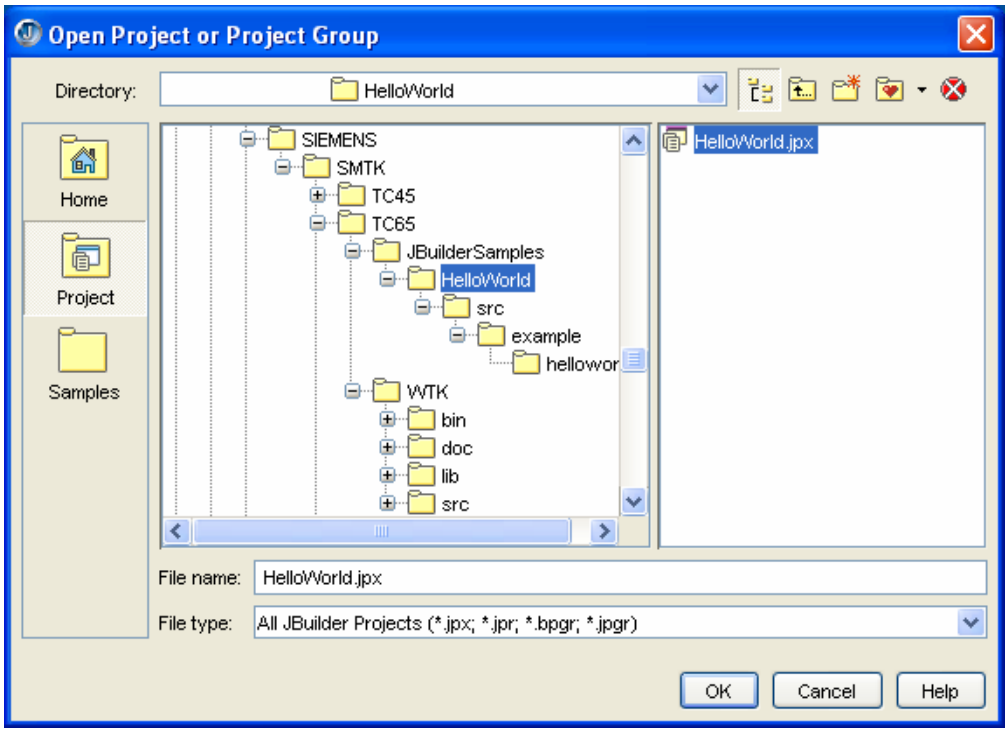


Figure 27: JBuilderX – Sample Projects

Open the Project (e.g. “HelloWorld.jpx”), rebuild the sources and start the debugger using the micro edition (context menu *“HelloWorld.jad”* → *Micro Debug using “HelloWorld”*).

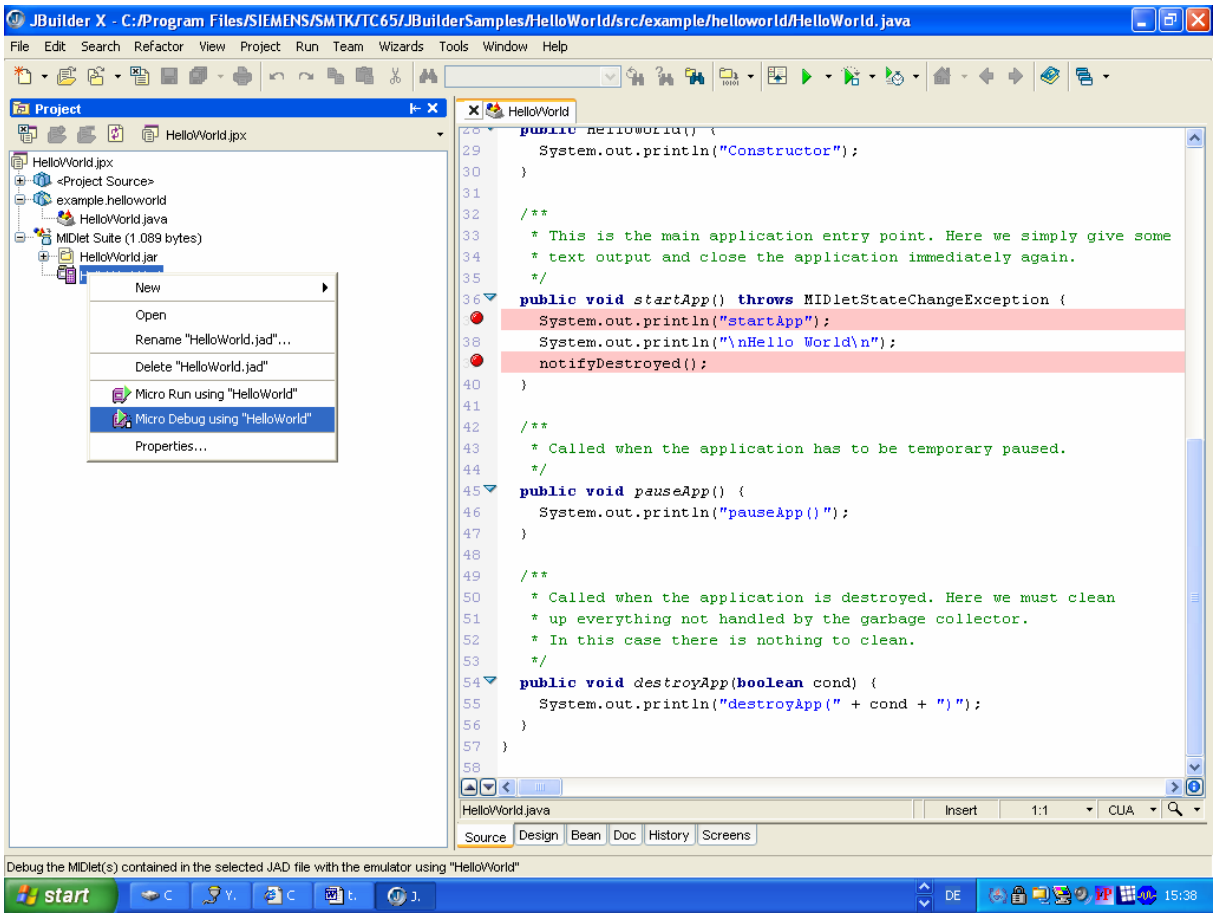


Figure 28: JBuilderX – Starting the debugging session

10.3.3 Borland JBuilder 2005

If JBuilder2005 is not installed, run the JBuilder2005 install program and follow the installation wizard instructions. Use the TC65 SMTK installation routine in maintenance mode. The SMTK will find the installed JBuilder2005 IDE. Select JBuilder2005 to integrate the SMTK into JBuilder2005.

After the integration of the TC65 SMTK into JBuilder2005 you can examine the integration by opening the menu *Tools -> Configure → JDKs…* (see Figure below)

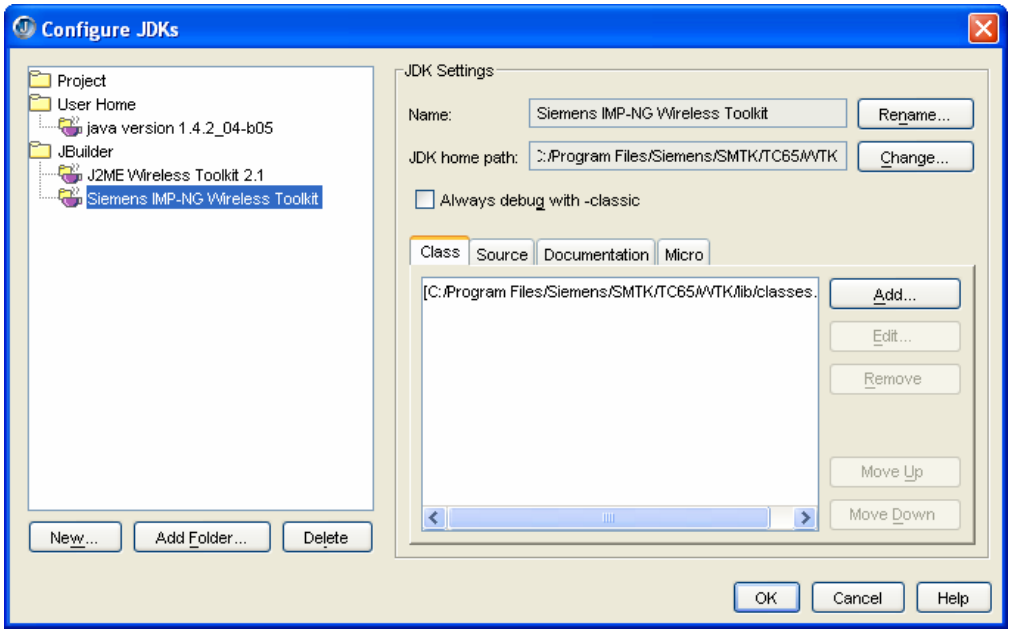


Figure 29: JBuilder2005 – JDK settings

The libraries included with the TC65 SMTK can be examined by opening the menu *Tools → Configure → Libraries…* (see Figure below)

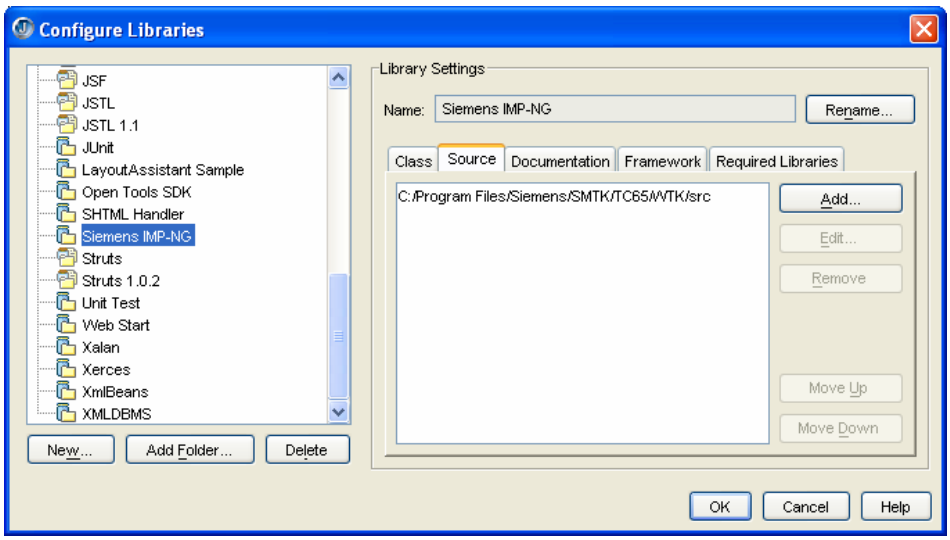


Figure 30: JBuilderX – Siemens Library

10.3.3.1 Examples

There are sample projects provided with the TC65 SMTK. These projects can be found in the JBuilderSamples directory of the TC65 SMTK installation directory. This directory is accessed by opening a project using the menu *File → Open Project…* (see Figure below)

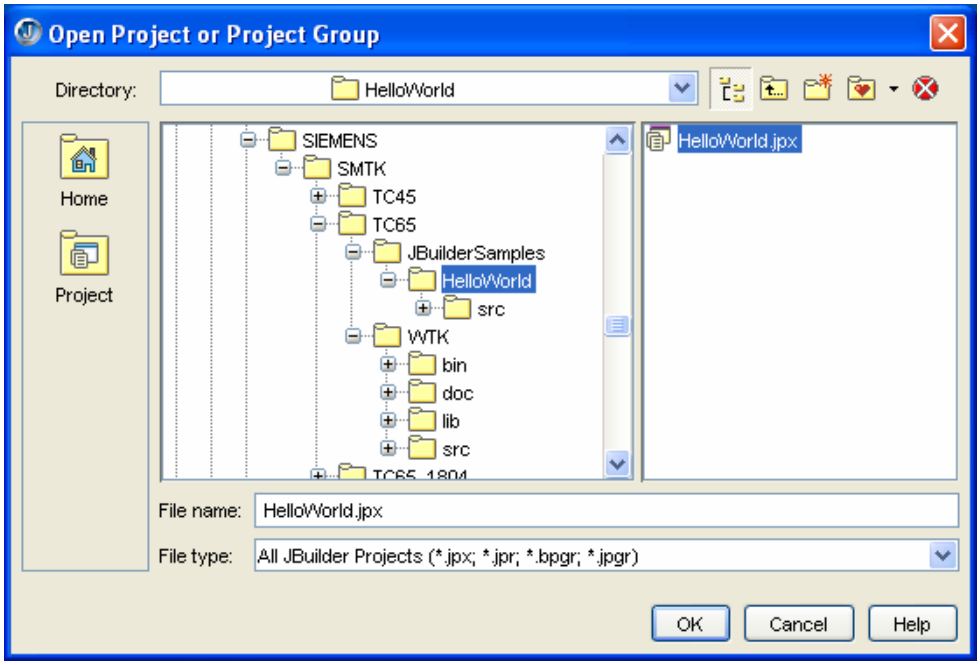


Figure 31: JBuilder2005 – Sample Projects

Starting the debugging session is done in the same way as for JBuilderX (see chapter for JBuilderX above)

10.3.4 Eclipse 3.0

If Eclipse is not installed, please unpack eclipse-SDK-3.0.2-win32.zip. We recommend that Eclipse 3.0.1 or higher be used.

10.3.4.1 Integration

Before you can run the installation routine of TC65 SMTK in maintenance mode, you need to install the EclipseME Plugin. This plug-in can normally be downloaded from http://eclipseme.org/. As long as the customizations necessary for TC65 are not part of an official release this customized version comes with the SMTK cd in the directory "EclipseMEplugin".

Start Eclipse and open the menu *Help→Software Updates→Find and Install…*
Select in the following window "*Search for new features to install*" and click *Next*

Now you need to select the *plugin.zip* after clicking the *New Archived Site…* button. (see figure below).

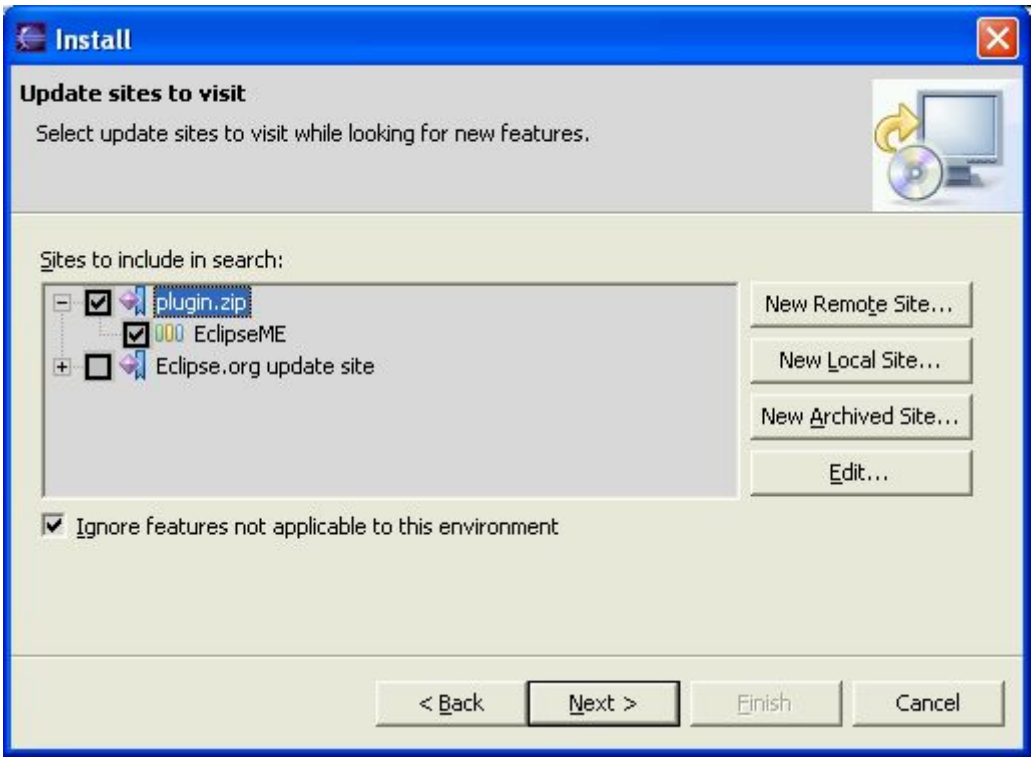


Figure 32: Eclipse – Plug-in installation

If the installation of the plugin was successful you need to restart your IDE. (see figure below).

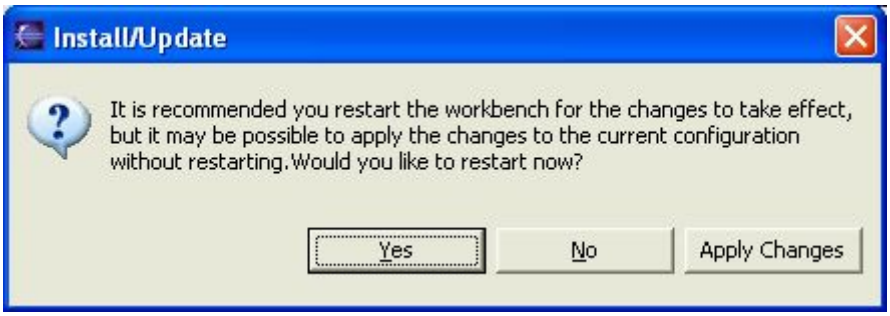

Install/Update
It is recommended you restart the workbench for the changes to take effect, but it may be possible to apply the changes to the current configuration without restarting. Would you like to restart now?
Yes
No
Apply Changes

Figure 33: Eclipse – Plug-in installation, restart

Close the IDE!

Now start the TC65 SMTK installation routine to automatically configure Eclipse. After the integration you can see the configuration in *Window→Preferences→J2ME→Platform Components* (see figure below)

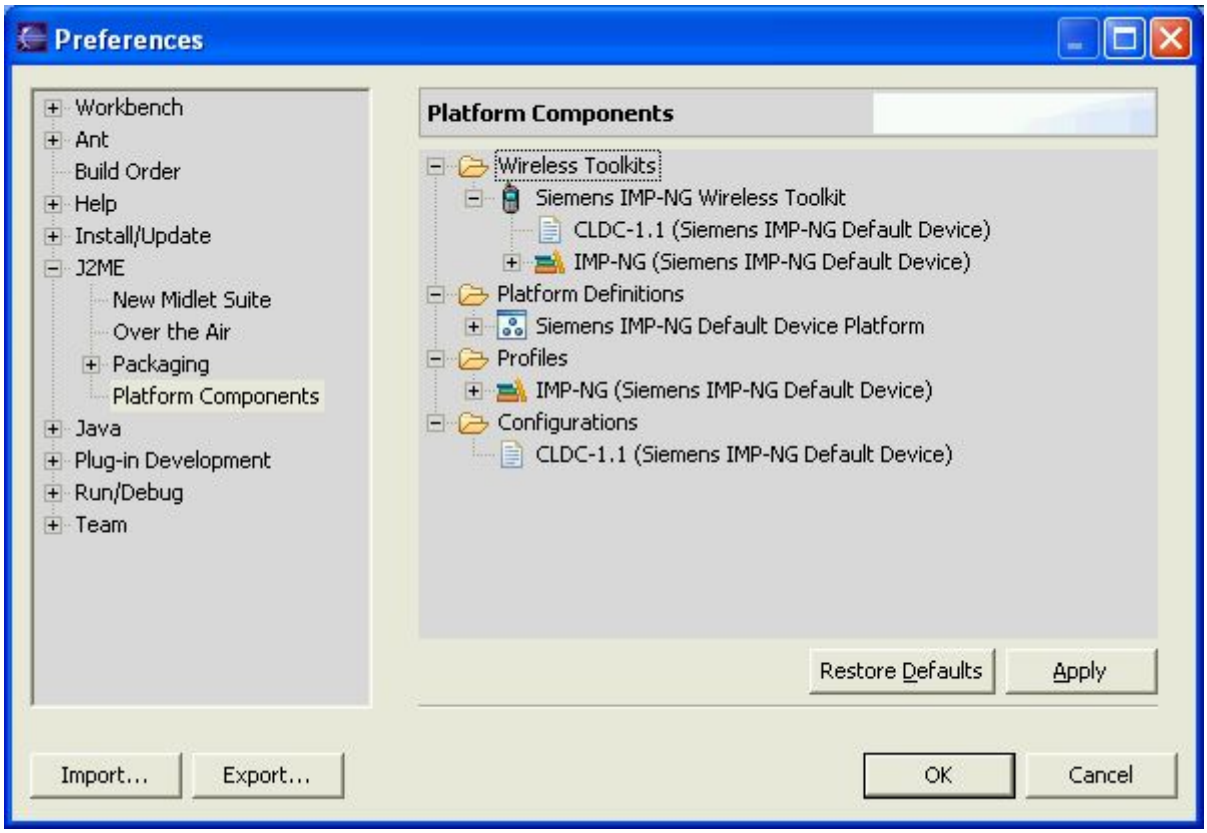


Figure 34: Eclipse – IMP-NG component

Set the timeout under Windows → Preferences →J2ME → Debug Server Delay to 15000. If you develop an extremely large application you may have to increase this timeout.

10.3.4.2 Switching Emulators

You can easily switch between installed SMTK by using the preferences of a project and choosing *IMP_NG_DefaultDevice*, see figure below.

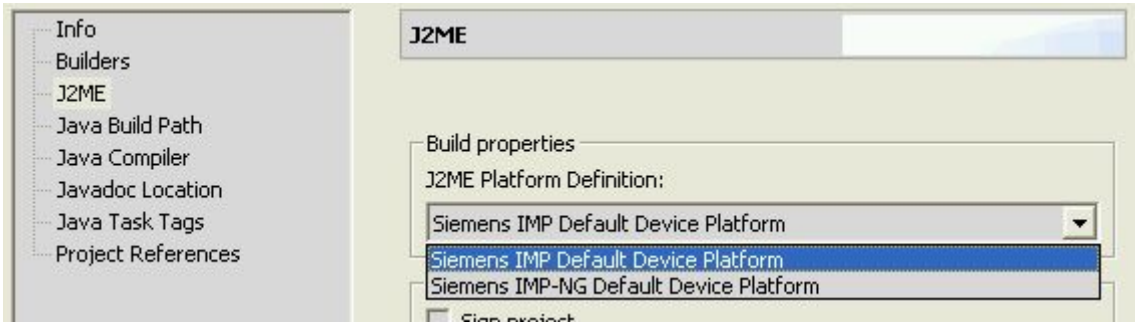


Figure 35: Eclipse – J2ME platform

10.3.4.3 Example

After the integration you can import one example. Open menu: *File→Import→Existing Project into Workspace* and choose the root directory of the example. (see figure below)

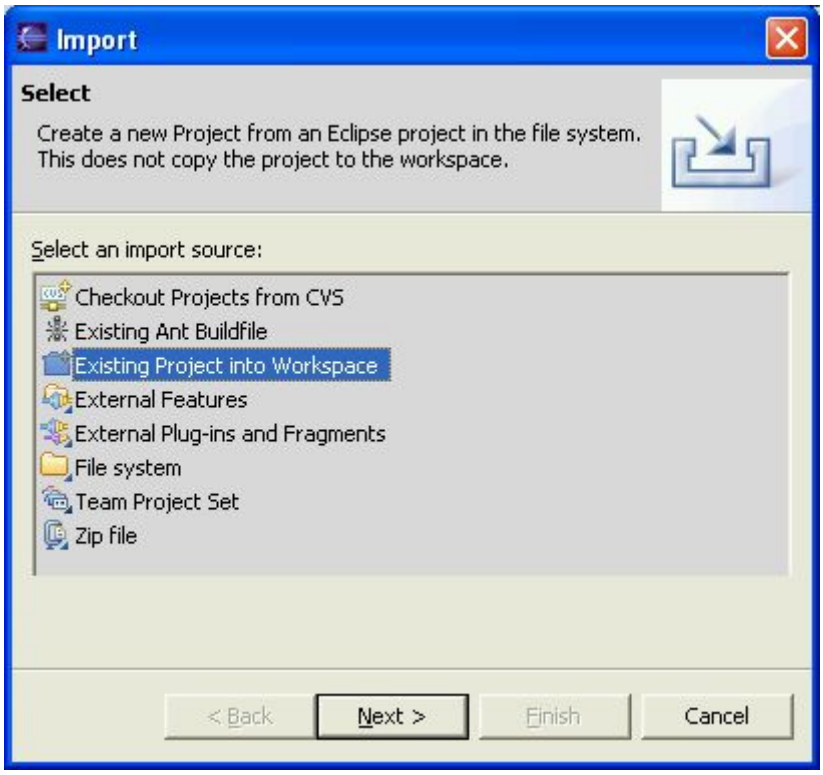


Figure 36: Eclipse – Project import

The following figure shows the “Hello World” example in the IDE.

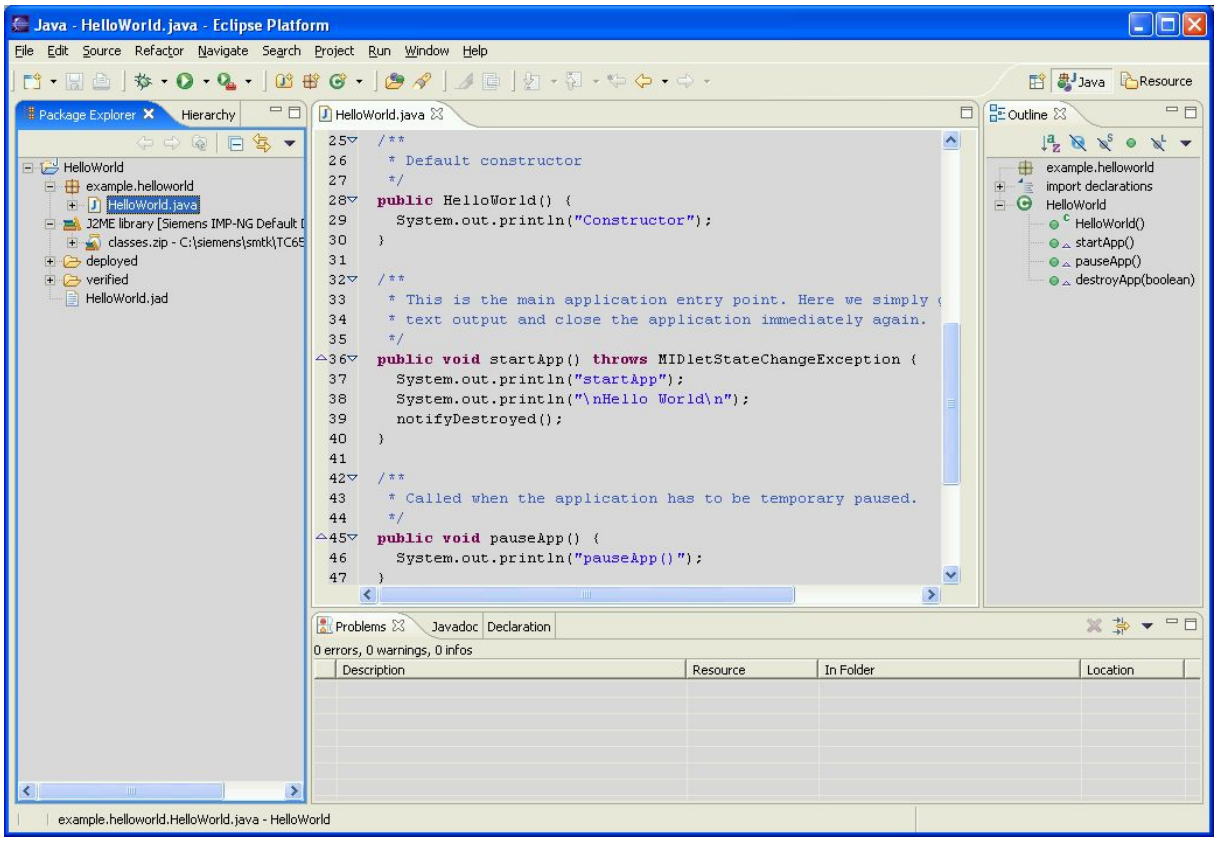


Figure 37: Eclipse - Example

The html help files of the SMTK can be accessed directly by pressing Alt+F2 while the cursor points to a Java expression in a Java source file.

10.3.4.4 Compile and debug

To build the jar and jad files you have rebuild the project with the “create package” function. Open the context menu of the project and go to *J2ME→Create Package*

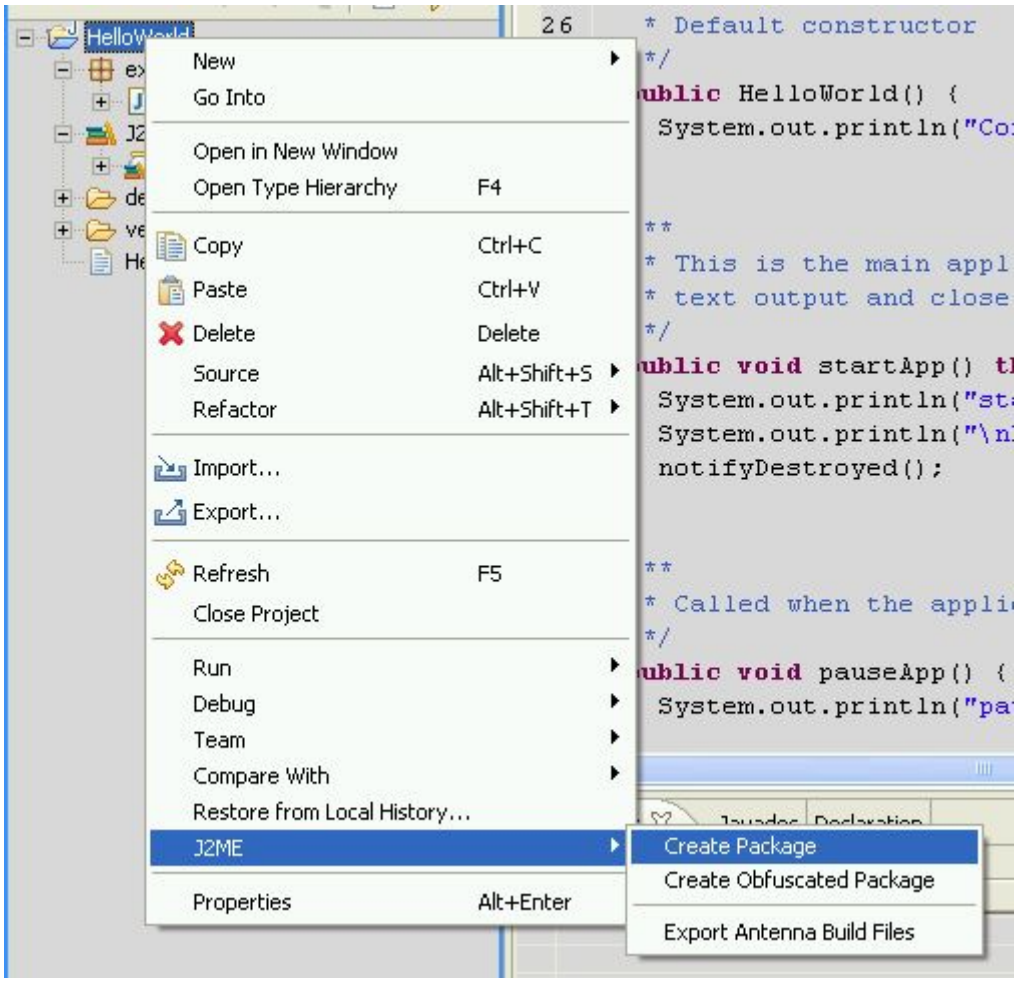


Figure 38: Eclipse – Create package

Now you are ready to debug your project. *Run→Debug…*
A TC65 launcher with pre settings is provided for starting the debugging session.
Please ensure that you have selected the right project and executable midlet.

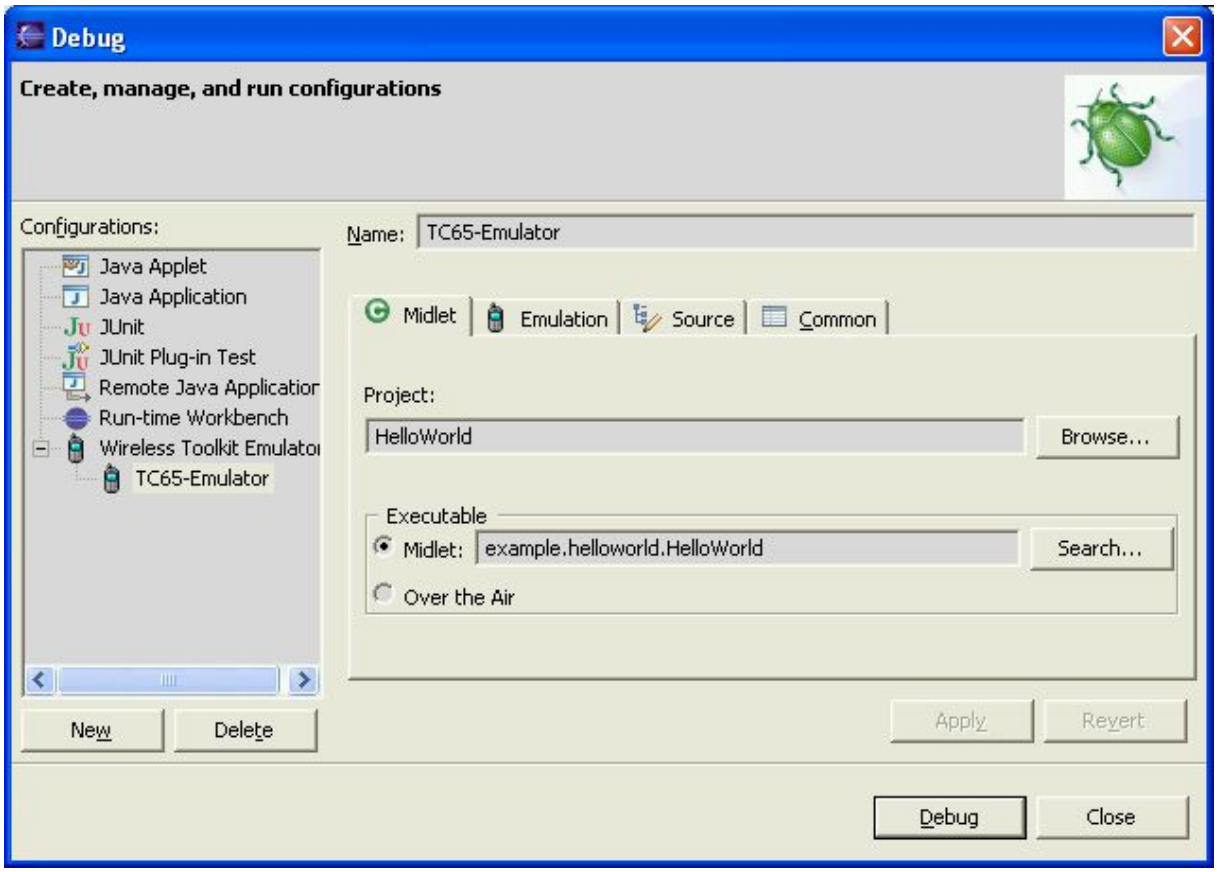


Figure 39: Eclipse - Configuration

Restriction: The integration of the SMTK depends selecting the offered default workspace “\workspace” of Eclipse.

10.4 Breakpoints

Breakpoints can be set as usual within the IDE. The debugger cannot step through methods or functions whose source code is not available.

11 Java Security

The Java Security Model follows the specification of IMP 2.0 but does not integrate the protection domain concept. The protection domain concept is not needed. If the software is enabled, all domains are available.

Java Security is divided into two main areas:

- secure MIDlet data links (HTTPS, Secure Connection) (see chapter 11.1)
- execution of signed MIDlets (see chapter 11.2 Execution Control)

11.1 Secure Data Transfer

This feature makes it possible for MIDlets to use safe data links to external communications partners. The specification IMP 2.0 defines two java classes with this characteristic - HTTPSConnection and SecureConnection.

The Siemens implementation follows the recommendations in IMP 2.0:

HTTPSConnection
- HTTP over TLS as documented in RFC 2818 and TLS Protocol Version 1.0 as specified in RFC 2246.

SecureConnection
- TLS Protocol Version 1.0 as specified in RFC 2246

Two Java Security modes exist for safe data links.
Mode 1:
- Java Security not activated
- No examination of the server certificate takes place when setting up the connection. The authenticity of the server certificate is not verified. (Figure 40: Mode 1 - Java Security not activated)

Mode 2 (see 11.2.1 Change to Secured Mode*):*
- The server certificate is examined when setting up a connection. Two configurations are valid. The server certificate is identical to the certificate in the module (both certificates are self signed root certificates) or the server certificate forms a chain with the certificate of the module. Thus the authenticity of the server certificate can be examined with the help of the certificate of the module. (Figure 41: Mode 2 - Java Security activated and
- Figure 42: Mode 2 - Java Security activated)

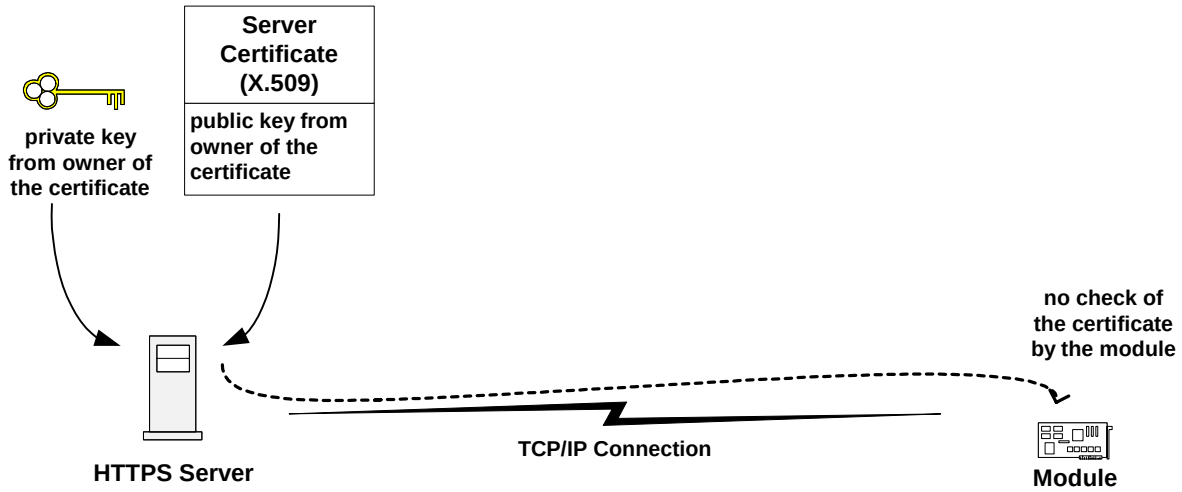

Server Certificate (X.509)
public key from owner of the certificate
private key from owner of the certificate
no check of the certificate by the module
TCP/IP Connection
HTTPS Server
Module

Figure 40: Mode 1 - Java Security not activated

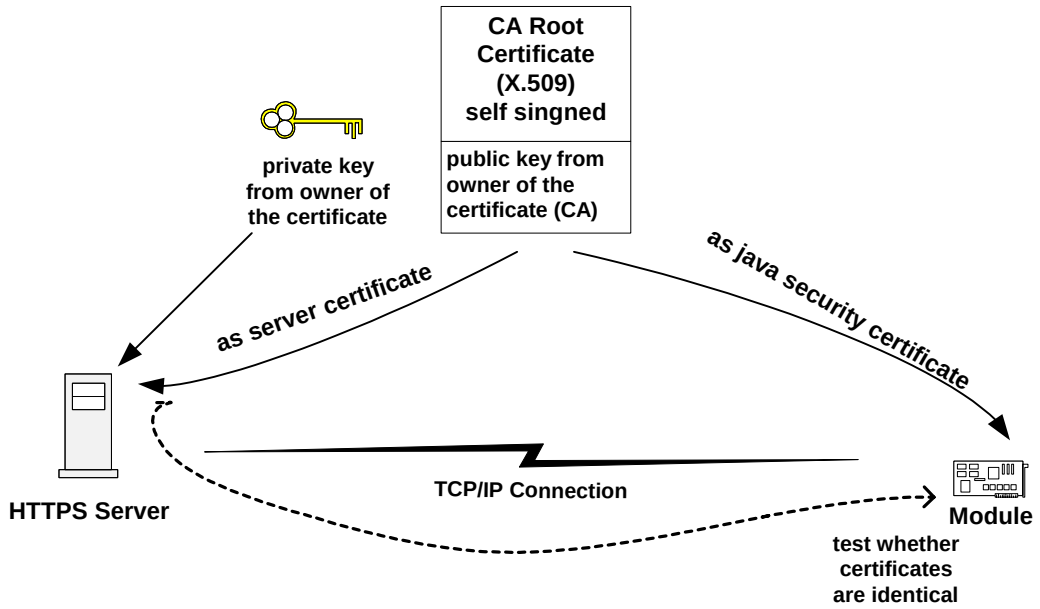

CA Root Certificate (X.509) self singned
public key from owner of the certificate (CA)
private key from owner of the certificate
as server certificate
as java security certificate
HTTPS Server
TCP/IP Connection
Module
test whether certificates are identical

Figure 41: Mode 2 - Java Security activated
(server certificate = certificate into module)

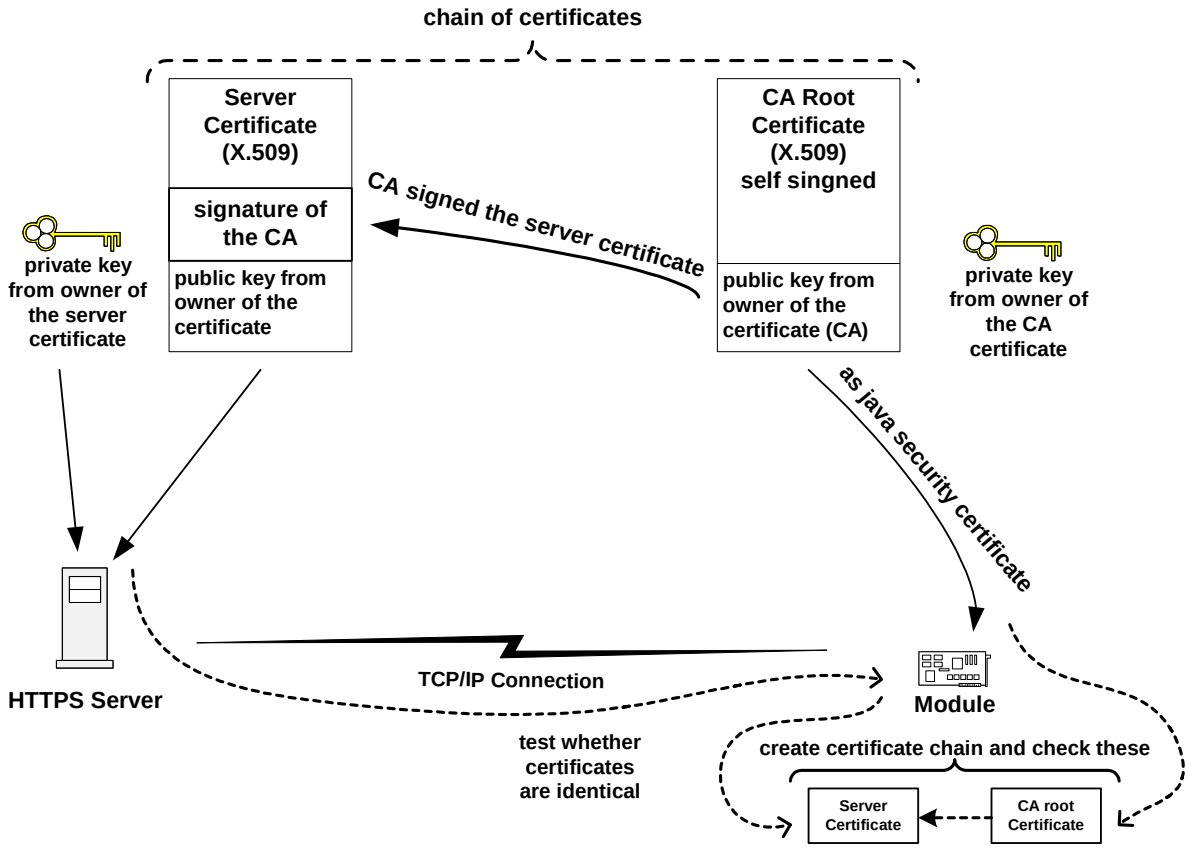

chain of certificates
Server Certificate (X.509)
signature of the CA
public key from owner of the certificate
CA signed the server certificate
CA Root Certificate (X.509) self singned
public key from owner of the certificate (CA)
private key from owner of the server certificate
private key from owner of the CA certificate
as java security certificate
HTTPS Server
TCP/IP Connection
Module
test whether certificates are identical
create certificate chain and check these
Server Certificate
CA root Certificate

Figure 42: Mode 2 - Java Security activated
(server certificate and self signed root certificate in module form a chain)

11.1.1 Create a Secure Data Transfer Environment Step by Step

The following steps describe the creation of the configuration:

- Java Security Mode is activated (*see* 11.2.1 Change to Secured Mode)
- Certificate verification is activated for a data connection (HTTPS or SecureConnection)

The steps described below use the cygwin + openssl environment (for installation see http://www.cygwin.com/, the openssl documentation can be found here http://www.openssl.org/docs/apps/openssl.html)

1. *Create CA and generate CA Root Certificate*
   - We need certificates with sha1 signature. Java Security supports a sha1 signature of the certificate only.
     Add the parameter "-sha1" to the command "Making CA certificate ..." in the section of file CA.pl (cygwin location *"\cygwin\usr\ssl\misc"*)
   - create a shell (use location *cygwin\usr\ssl\misc*)
   - execute commands
     >perl CA.pl –newca
   - convert file format from PEM to DER
     CA certificate cacert.pem
     >**openssl x509 -in ./demoCA/cacert.pem -inform PEM**
     -out ./demoCA/cacert.der -outform DER
     CA private key file cakey.pem
     >**openssl pkcs8 -in ./demoCA/private/cakey.pem**
     -inform PEM
     -out ./demoCA/private/cakey.der
     -outform DER -nocrypt -topk8

2. *Create server certificate and java keystore*
   - execute command
     >keytool -genkey -alias server -keypass keypass
     -keystore customer.ks -storepass keystorepass
     -sigalg SHA1withRSA -keyalg RSA
     the field "name" of the certificate is the domain name or the IP address of the server

3. *Create certificate request for server certificate*
   - execute command
     >keytool -certreq -alias server -file server.csr
     -keypass keypass
     -keystore customer.ks -storepass keystorepass

4. *Sign certificate request by CA*
   - execute command
     >openssl ca -in server.csr -out server.pem
   - convert file format from PEM to DER
     >openssl x509 -in server.pem -inform PEM
     –out server.der -outform DER

5. *Import CA root certificate and CA private key into java keystore*
   - Use the CA Root Certificate for the creation of Java Security Command (see chapter 11.5.3)
   - execute command
     >java -jar setprivatekey.jar -alias dummyca
     -storepass keystorepass -keystore customer.ks
     -keypass cakeypass

```
                    -keyfile ./democa/private/cakey.der
                    -certfile ./democa/cacert.der
```

6. Export private key from server certificate

- The private key is needed for the (HTTPS or Secure Connection)server configuration.
- execute command

```
>java -jar getprivatekey.jar -alias server
          -keystore customer.ks -storepass keystorepass
          -keypass keypass -keyfile server_privkey.der
```

Result:

- You have a keystore for the configuration of the Java Security of the module
- You have a signed server certificate (files ““server.pem” or “server.der”)
- You have a private key file for your server configuration

11.2 Execution Control

The Java environment of the TC65 module has two modes.

unsecured mode:

- The device starts all java applications (MIDlets).

secured mode

- The customer can activate the secured mode of the device. For this the customer sends a root certificate (x.509 certificate) to the device (over an AT-Interface). The device changes from “unsecured mode” to the “secured mode”. From this time the module will only start java applications with a valid signature. In addition, the device will only accept special commands from the customer if they are marked with a signature. The device examines each command with the public key of the customer root certificate

The secured mode is activated by a special AT-command.
Siemens supplies modules with unsecured mode as the default configuration.

11.2.1 Change to Secured Mode Concept

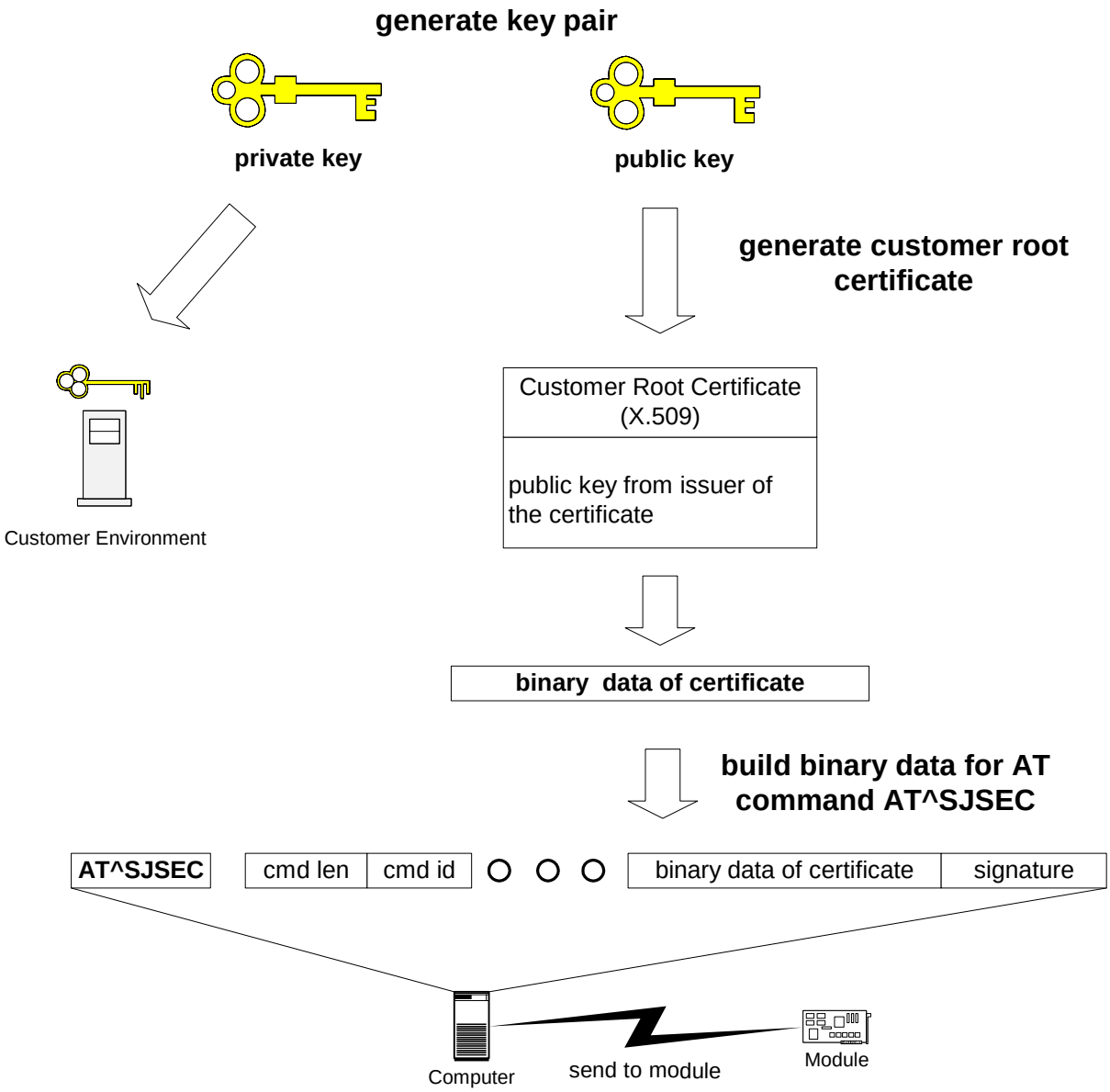


Figure 43: Switch to Security Mode

11.2.2 Concept for the Signing the Java MIDlet

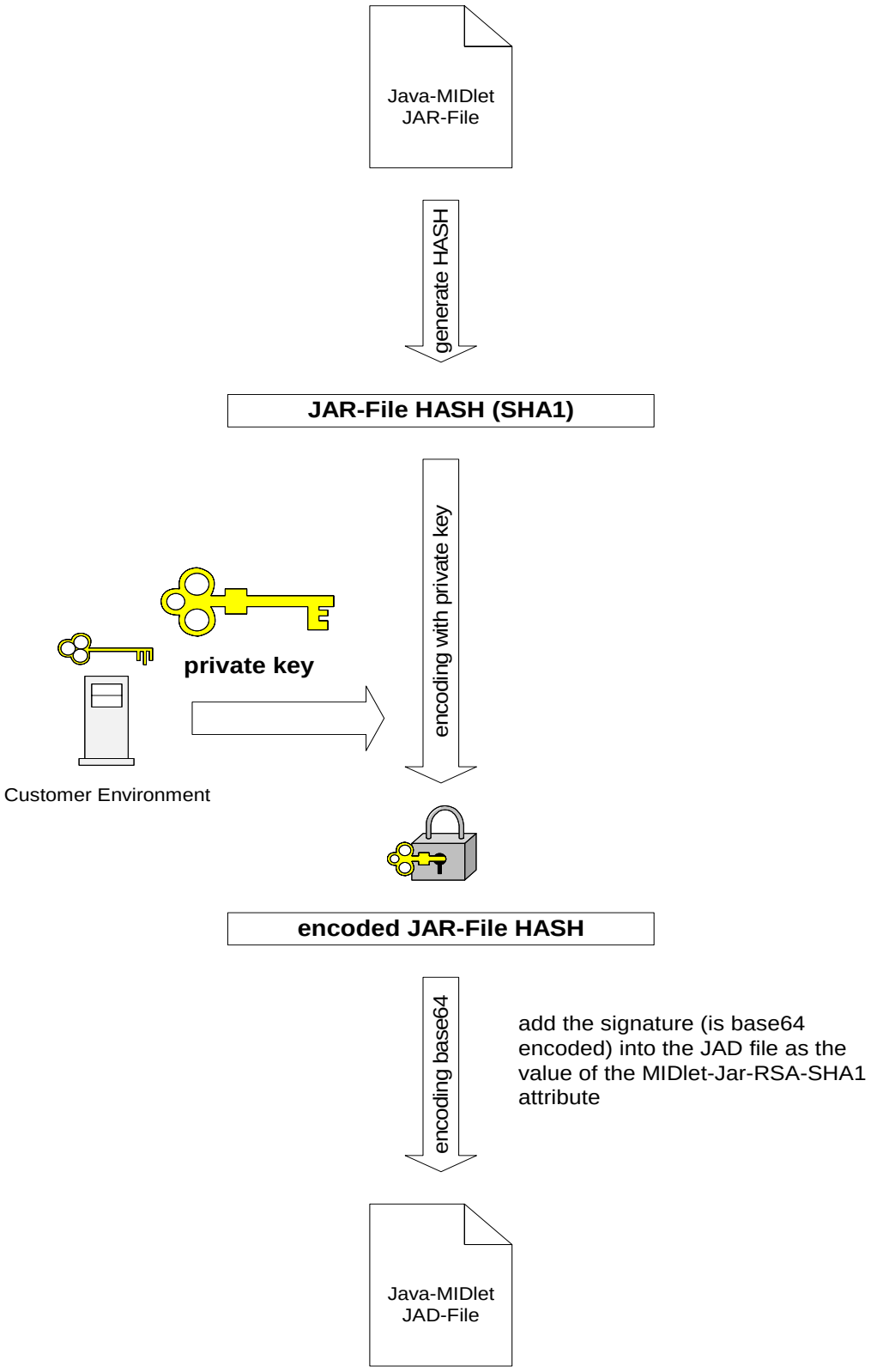


Figure 44: Prepare MIDlet for Secured Mode

11.3 Application and Data Protection

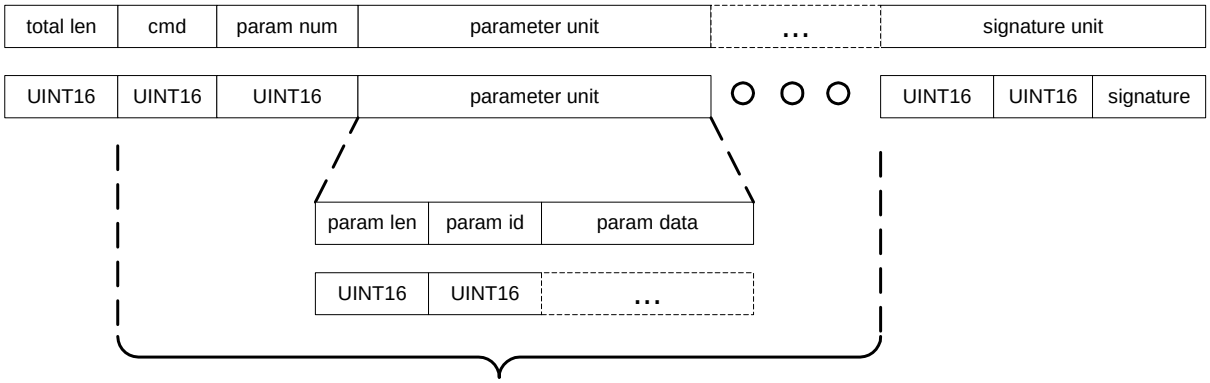
In addition to the Java secured mode it is possible to prevent the activation of the Module Exchange Suite. When Module Exchange Suite access is deactivated with *at^sjsec*, it is no longer possible to access to the Flash file system on the module.

11.4 Structure and Description of the Java Security Commands

Special commands are used in t he Java security environment. These commands are transferred to the module with the help of the special AT command *at^sjec*. This command makes it possible to send binary data to the module. After *at^sjsec* is issued, the module changes into a block transfer mode. Now binary data in a fixed format can be sent. These binary data contain the actual Java security commands.

11.4.1 Structure of the Java Security Commands

General structure



total len = all bytes of the command structure (including size of "total len")

param len = all bytes of the parameter structure (including size of "param len")

List of parameters

param id	param len	param data	description
0x0001	D	certificate data	content of *.der file
0x0002	0x0005	0x00 or 0x01	on/off switch, 0x00 = off, 0x01 = on
0x0003	0x0014	IMEI	numeric numbers in ASCII format (zero terminated string)
0x0004	D	signature data	SHA-1signature the of command, base64 coded (zero terminated string)

D - depend from the length of parameter

List of commands

cmd id	description
0x0001	Set Customer Root Certificate
0x0002	Del Customer Root Certificate
0x0003	Switch on/off Certificate Verification for HTTPS Connections
0x0004	Switch on/off OBEX Functionality

Set Customer Root Certificate

total len	0x0001	0x0003	param unit certificate	param unit IMEI	param unit signature

Del Customer Root Certificate

total len	0x0002	0x0002	param unit IMEI	param unit signature

Switch on/off Certificate Verification for HTTPS Connections

total len	0x0003	0x0003	param unit switch	param unit IMEI	param unit signature

Switch on/off OBEX Functionality

total len	0x0004	0x0003	param unit switch	param unit IMEI	param unit signature

11.4.2 Build Java Security Command

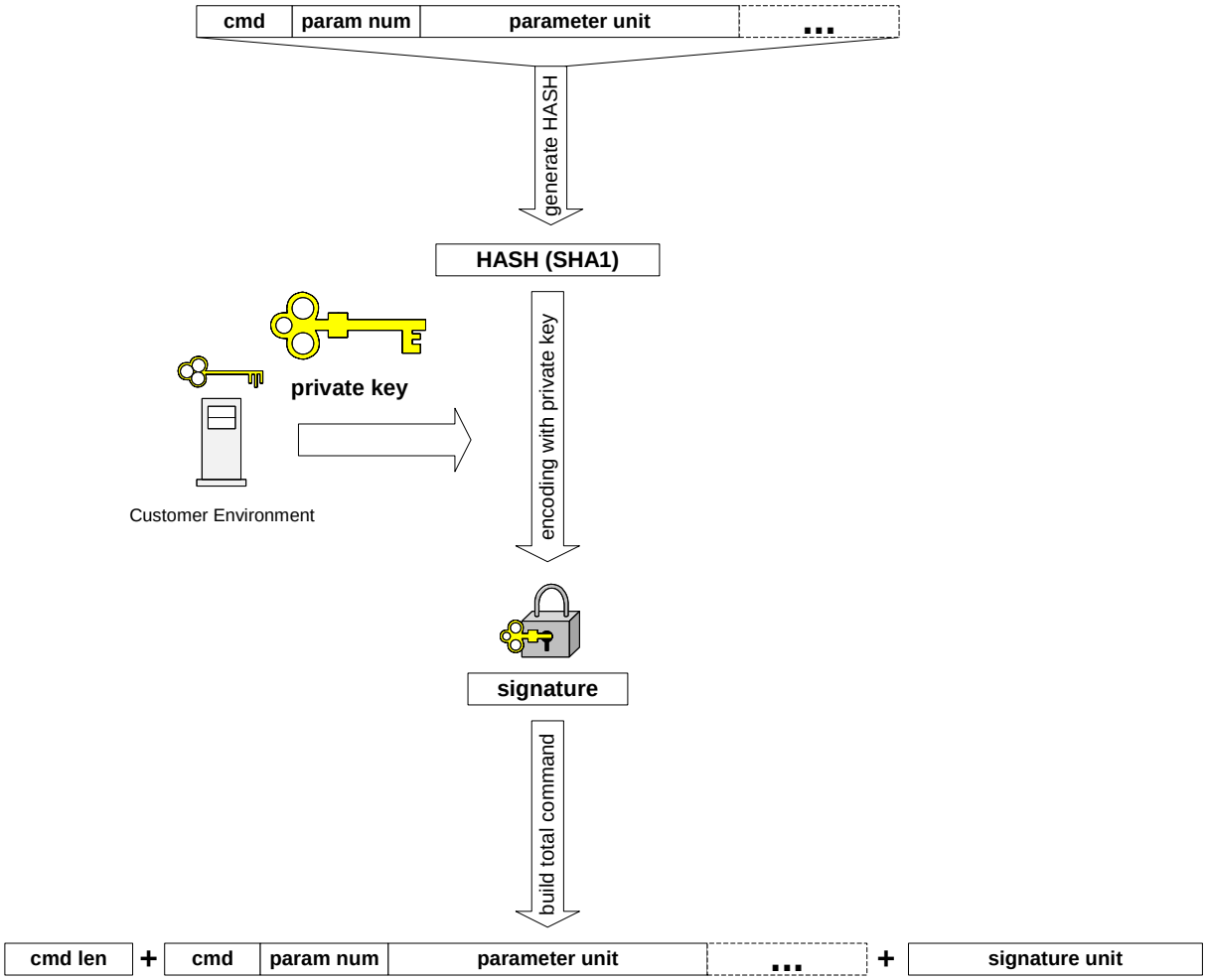


Figure 45: Build Java Security Command

11.4.3 Send Java Security Command to the Module

Use a terminal program.
enter:

```
at^sjsec
```

wait for the answer:

```
CONNECT

JSEC READY: SEND COMMAND ...
```

Now you can send the binary data of the command (for example: from a file with the binary data of the command).
The module's answer depends on the result of the operation.

The read command, *at^sjsec?*, can be used to request the current Java security status.

Read command AT^SJSEC?	The read command can be used to request the current status of java security. Response ^SJSEC:<state>[,<HTTPS state>,<OBEX state>,<certificate content>] <state> java security mode 0 security mode not active, no information follows 1 security mode active, further information follows <OBEX state>,<HTTPS state>,<certificate information> <HTTPS state> 0 the HTTPS connection or Secure Connection is possible if the server certificate (or certificate chain) is valid 1 the HTTPS Connection or Secure Connection is possible only if the server certificate is signed by the customer (owner of root certificate in device) <OBEX state> 0 start of Module Exchange Suite is not permitted 1 start of Module Exchange Suite is permitted Certificate Information: Issuer: SerialNumber: Subject: Signature algorithm: Thumbprint algoritm: Thumbprint:

11.5 Create a Java Security Environment Step by Step

11.5.1 Create Key store

The key store contains the key pairs for signing data. For producing the key store with keys the tool “keytool.exe” can be used.

The program is in the Java SDK. (for a description see http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html)

Example:

```
keytool -genkey -alias keyname -keypass keypassword -keystore customer.ks
        -storepass keystorepassword -sigalg SHA1withRSA -keyalg RSA
```

11.5.2 Export X.509 Root Certificate

For exporting the x.509 root certificate use “keytool.exe”. The program is in the Java SDK. (for description see http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/keytool.html)

```
keytool -export -v -keystore customer.ks -storepass keystorepassword
        -alias keyname > certificate.der
```

11.5.3 Create Java Security Commands

For producing the java security commands the tool “jseccmd.jar” can be used.
This program is in the folder “wkt\bin”.

Command: switch to java security mode (Set Customer Root Certificate)

```
java -jar jseccmd.jar -cmd SetRootCert -certfile customer.der
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > SetRootCert.bin
```

Command: switch to java normal mode (Del Customer Root Certificate)

```
java -jar jseccmd.jar -cmd DelRootCert
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > DelRootCert.bin
```

Command: switch on Certificate Verification for HTTPS Connections

```
java -jar jseccmd.jar -cmd HttpsVerifyOn
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > HttpsVerifyOn.bin
```

Command: switch off Certificate Verification for HTTPS Connections

```
java -jar jseccmd.jar -cmd HttpsVerifyOff
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > HttpsVerifyOff.bin
```

Command: switch on module exchange functionality

```
java -jar jseccmd.jar -cmd ObexActivationOn
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > ObexActivationOn.bin
```

Command: switch off module exchange functionality

```
java -jar jseccmd.jar -cmd ObexActivationOff
                      –imei 012345678901234 -alias keyname
                      -storepass keystorepassword -keypass keypassword
                      -keystore customer.ks > ObexActivationOff.bin
```

11.5.4 Sign a MIDlet

Use the tool “jadtool.jar” to sign a Java MIDlet. This program is in the folder “wkt\bin”.

```
java -jar jadtool.jar -addjarsig -jarfile helloworld.jar
                                 -inputjad helloworld.jad
                                 -outputjad helloworld.jad
                                 -alias keyname -storepass keystorepassword
                                 -keypass keypassword -keystore customer.ks
                                 -encoding UTF-8
```

11.6 Attention

The central element of Java Security is the **private key**. If Java Security is activated and you lose the private key, then the **module is destroyed**. You do not have a possibility of deactivating Java Security, downloading of a new Midlet or of starting any other operation concerning Java Security.
In order to prevent this, we recommend to you particularly to **secure the private key**.

12 Java Tutorial

This small tutorial includes explanations on how to use the AT Command API and suggestions for programming MIDlets. The developer should read about MIDlets, Threads and AT commands as a complement to this tutorial.

12.1 Using the AT Command API

Perhaps the most important API for the developer is the AT command API. This is the API that lets the developer issue commands to control the module. This API consists of the *ATCommand* class and the *ATCommandListener* and *ATCommandResponseListener* interfaces. Their javadocs can be found in *…\wtk\doc\html\index.html*, [5].

12.1.1 Class ATCommand

The *ATCommand* class supports the execution of AT commands in much the same way as they would be executed over a serial interface. It provides a simple way to send strings directly to the device's AT parsers.

12.1.1.1 Instantiation with or without CSD Support

There can be only exactly as many *ATCommand* instances as there are parsers on the device. If there are no more parsers available, the *ATCommand* constructor will throw *ATCommandFailedException*. All AT parser instances support CSD. However from a Java application point of view it may make sense to have one dedicated instance for CSD call handling. Therefore, and also for historical reasons, only one parser with CSD support may be requested through the constructor. If more then one parser with CSD support is requested, the constructor will throw *ATCommandFailedException*.

```
try {
  ATCommand atc = new ATCommand(false);
  /* An instance of ATCommand is created. CSD is not explicitly
   * requested. */
} catch (ATCommandFailedException e) {
  System.out.println(e);
}
```

The csdSupported() method returns the CSD capability of the connected instance of the device's AT parser.

```
boolean csd_support = atc.csdSupported();
```

release() releases the resources held by the instance of the ATCommand class. After calling this function the class instance cannot be used any more but the resources are free to be used by a new instance

12.1.1.2 Sending an AT Command to the Device, the send() Method

An AT command is sent to the device by using the send() method. The AT command is sent as a string which must include the finalizing line feed "\r" or the corresponding line end character.

```
String response = atc.send(“at+cpin?\r”);
/* method returns when the module returns a response */
System.out.println(response);
```

Possible response printed to System.out:
```
+CPIN: READY OK
```

This send function is a blocking call, which means that the calling thread will be interrupted until the module returns a response. The function returns the response, the result code of the AT command, as a string.

Occasionally it may be infeasible to wait for an AT command that requires some time to be processed, such as **at+cops?**. There is a second, non-blocking, send function which takes a second parameter in addition to the AT command. This second parameter is a callback instance, *ATCommandResponseListener*. Any response to the AT command is delivered to the callback instance when it becomes available. The method itself returns immediately. The *ATCommandResponseListener* and the non-blocking send method are described in Section 12.1.2.

Note: Using the send methods with strings with incorrect AT command syntax will cause errors.

12.1.1.3 Data Connections

If a data connection is created with the *ATCommand* class, for instance with **‘***atd***’**, an input stream is opened to receive the data from the connection. Similarly, an output stream can be opened to send data on the connection.

```
/* Please note that this example would not work unless the module had
 * been initialized and logged into a network. */

System.out.println("Dialing: ATD" + CALLED_NO);
response = atc.send("ATD" + CALLED_NO + "\r");
System.out.println("received: " + response);

if (response.indexOf("CONNECT") >= 0) {
  try {
    // We have a data connection, now we do some streaming...
    // IOException will be thrown if any of the Stream methods fail
    OutputStream dataOut = ATCmd.getDataOutputStream();
    InputStream dataIn = ATCmd.getDataInputStream();

    // out streaming...
    dataOut.write(new String("\n\rHello world\n\r").getBytes());
    dataOut.write(new String("\n\rThis data was sent by a Java " +
                             "MIDlet!\n\r").getBytes());
    dataOut.write(new String("Press 'Q' to close the " +
                             "connection\n\r").getBytes());

    // ...and in streaming
    System.out.println("Waiting for incoming data, currently " +
                       dataIn.available() + " bytes in buffer.");
    rcv = 0;
```

```
    while(((char)rcv != 'q') && ((char)rcv != 'Q') && (rcv != -1)){
      rcv = dataIn.read();
      if (rcv >= 0) {
        System.out.print((char)rcv);
      }
    }

    /* The example continues after the next block of text */
```

In **…/Siemens/SMTK/TC65/wtk/src/example** a complete data connection example, DataConnectionDemo.java, can be found.

These streams behave slightly differently than regular data streams. The streams are not closed by using the close() method. A stream remains open until the *release()* method is called. A module can be switched from the data mode to the AT command mode by calling the *breakConnection()* method.

```
    /* continue example */

    if (rcv != -1) {
      // Now break the data connection
      System.out.println("\n\n\rBreaking connection");
      try {
        strRcv = ATCmd.breakConnection();
      } catch(Exception e) {
        System.out.println(e);
      }
      System.out.println("received: " + strRcv);
    } else {
      // Received EOF, somebody else broke the connection
      System.out.println("\n\n\rSomebody else switched to " +
                         "command mode!");
    }
    System.out.println("Hanging up");
    strRcv = ATCmd.send("ATH\r");
    System.out.println("received: " + strRcv);
  } catch(IOException e) {
    System.out.println(e);
  }
} else {
  System.out.println("No data connection established,");
}
```

An IOException is thrown if any function of the I/O streams are called when the module is in AT command mode.

The ATCommand class does not report the result codes returned after data connection release.

Data Connections are not only used for data transfer over the air but also to access external hardware. Here is a list of at commands which open a data connection:

- atd, for data calls
- at^sspi, for access to I2C/SPI
- at^sis commands for Internet services

For data connection signaling see also chapter 12.1.3

12.1.1.4 Synchronization

For performance reasons no synchronization is done in the ATCommand class. If an instance of this class has to be accessed from different threads ensure that the *send()* functions, the *release()* function, the *cancelCommand()* function and the *breakConnection()* function are synchronized in the user implementation.

12.1.2 ATCommandResponseListener Interface

The *ATCommandResponseListener* interface defines the capabilities for receiving the response to an AT command sent to one of the module's AT parsers. When the user wants to use the non blocking version of the *ATCommand.send* function of an implementation class of the *ATCommandResponseListener* interface must be created first. The single method of this class, *ATResponse()*, must contain the processing code for the possible response to the AT command sent.

```
class MyListener implements ATCommandResponseListener {

  String listen_for;

  public MyListener(String awaited_response) {
    listen_for = awaited_response;
  }

  void ATResponse(String Response) {
    if (Response.indexOf(listen_for) >= 0) {
      System.out.println("received: " + strRcv);
    }
  }
}
```

12.1.2.1 Non-blocking ATCommand.send() Method

After creating an instance of the *ATCommandResponseListener* class, the class instance can be passed as the second parameter of the non-blocking ATCommand.send() method. After the AT command has been passed to the AT parser, the function returns immediately and the response to the AT command is passed to this callback class later when it becomes available

Somewhere in the application:

```
MyListener connect_list = new MyListener("CONNECT");
atc.send("ATD" + CALLED_NO + "\r", connect_list);

/* Application continues while the AT command is processed*/
/* When the module delivers its response to the AT command the callback
 * method ATResponse is called. If the response is "CONNECT", we will see
 * the printed message from ATResponse in MyListener. */
```

A running AT command sent with the non-blocking send function can be cancelled with *ATCommand.cancelCommand()***.** Any possible responses to the cancellation are sent to the waiting callback instance.

Note: Using the send methods with incorrect AT command syntax in the strings will cause errors.

12.1.3 ATCommandListener Interface

The *ATCommandListener* interface implements callback functions for:

- URCs
- Changes of the serial interface signals RING, DCD and DSR
- Opening and closing of data connections

The user must create an implementation class for *ATCommandListener* to receive AT events. The *ATEvent* method of this class must contain the processing code for the different AT-Events (URCs). The *RINGChanged, DCDChanged, DSRChanged* and *CONNChanged* methods should contain the processing code for possible state changes.

12.1.3.1 ATEvents

An ATEvent or a URC is a report message sent from the module to the application. An unsolicited result code is either delivered automatically when an event occurs or as a result of a query the module previously received. However, a URC is not issued as a *direct* response to an executed AT command. Some URCs must be activated with an AT command.

Typical URCs may be information about incoming calls, a received SM, temperature changes, the status of the battery, etc. A summary of URCs is listed in the AT Command Set document [3].

12.1.3.2 Implementation

```
class ATListenerA implements ATCommandListener {

public void ATEvent(String Event) {
  if (Event.indexOf("+CALA: Reminder 1") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 2") >= 0) {
    /* take desired action after receiving the reminder */
  } else if (Event.indexOf("+CALA: Reminder 3") >= 0) {
    /* take desired action after receiving the reminder */
  }

  /* No action taken for these events */
  public void RINGChanged(boolean SignalState) {}
  public void DCDChanged(boolean SignalState) {}
  public void DSRChanged(boolean SignalState) {}
}

class ATListenerB implements ATCommandListener {

  public void ATEvent(String Event) {
    if (Event.indexOf("+SCKS: 0") >= 0) {
      System.out.println("SIM Card is not inserted.");
      /* perform other actions */
    } else if (Event.indexOf("+SCKS: 1") >= 0) {
      System.out.println("SIM Card is inserted.");
     /* perform other actions */
    }
  }

  public void RINGChanged(boolean SignalState) {
    /* take some action when the RING signal changes if you want to */
  }

  public void DCDChanged(boolean SignalState) {
```

```
    /* take some action when the DCD signal changes if you want to */
  }

  public void DSRChanged(boolean SignalState {}
    /* take some action when the DSR signal changes if you want to */
  }

  public void CONNChanged(boolean SignalState {}
    /* take some action when the state of a connection changes if you want
    to */
  }

}
```

12.1.3.3 Registering a Listener with an ATCommand Instance

After creating an instance of the *ATCommandListener* class, it must be passed as a parameter to the *ATCommand.addListener()* method. The callback methods of the instance will be called by the runtime system each time the corresponding events (URCs or signal state changes) occur on the corresponding device AT parser.

```
/* we have two ATCommands instances, atc1 and atc2 */
ATListenerA reminder_listener = new ATListenerA();
ATListenerB card_listener = new ATListenerB();

atc1.addListener(reminder_listener);
atc2.addListener(card_listener);
```

The *ATCommand.removeListener()* method removes a listener object that has been previously added to the internal list table of listener objects. After it has been removed from the list it will not be called when URCs occur. If it was not previously registered the list remains unchanged.

The same *ATCommandListener* may be added to several *ATCommand* instances and several *ATCommandListeners* may be added to the same *ATCommand.*

12.2 Programming the MIDlet

The life cycle and structure of MIDlets are described in Chapter 6. Since the MIDlets will run on J2ME™, all of J2ME™'s features, including threads, are available. Small applications, such as those without any timer functions or those used only for tests and simple examples, can be written without using threads. Longer applications should be implemented with threads.

12.2.1 Threads

Although small applications can be written without using threads longer applications should use them. The Java programming language is naturally multi-threaded which can make a substantial difference in the performance of your application. Therefore we recommend referring to Java descriptions on threads before making any choices about threading models. Threads can be created in two ways. A class can be a subclass of *Thread* or it can implement *Runnable*.

For example, threads can be launched in *startApp()* and destroyed in *destroyApp().* Note that destroying Java threads can be tricky. It is recommended that the developer read the Java documentation on threads. It may be necessary to poll a variable within the thread to see if it is still alive.

12.2.2 Example

/* This example derives a class from Thread and creates two instances
 * of the subclass. One thread instance finishes itself, the other one
 * is stopped by the main application. */

```
package example.threaddemo;

import javax.microedition.midlet.*;

public class ThreadDemo extends MIDlet {

  /* Member variables */
  boolean    runThreads = true; // Flag for stopping threads
  DemoThread thread1;           // First instance of DemoThread
  DemoThread thread2;           // Second instance of DemoThread

  /* Private class implementing the thread to be started by the
   * main application */
  private class DemoThread extends Thread {
    int loops;

    public DemoThread(int waitTime) {
      /* Store number of loops to execute */
      loops = waitTime;
      System.out.println("Thread(" + loops + "): Created");
    }

    public void run() {
      System.out.println("Thread(" + loops + "): Started");
      for (int i = 1; i <= loops; i++) {
        /* Check if main application asked thread to die */
        if (runThreads != true) {
          System.out.println("Thread(" + loops + "): Stopped from outside");
          /* Leave thread */
          return;
        }
        /* Print loop counter and wait 1 second,
         * do something useful here instead */
        System.out.println("Thread(" + loops + "): Loop " + i);
```

```
        try {
          Thread.sleep(1000);
        } catch(InterruptedException e) {
          System.out.println(e);
        }
      }
      System.out.println("Thread(" + loops + "): Finished naturally");
    }
  }

  /**
   * ThreadDemo - constructor
   */
  public ThreadDemo() {
    System.out.println("ThreadDemo: Constructor, creating threads");
    thread1 = new DemoThread(2);
    thread2 = new DemoThread(6);
  }

  /**
   * startApp()
   */
  public void startApp() throws MIDletStateChangeException {
    System.out.println("ThreadDemo: startApp, starting threads");
    thread1.start();
    thread2.start();
    System.out.println("ThreadDemo: Waiting 4 seconds before stopping threads");
    try {
      Thread.sleep(4000);
    } catch(InterruptedException e) {
      System.out.println(e);
    }
    destroyApp(true);
    System.out.println("ThreadDemo: Closing application");
    notifyDestroyed();
  }

  /**
   * pauseApp()
   */
  public void pauseApp() {
    System.out.println("ThreadDemo: pauseApp()");
  }

  /**
   * destroyApp()
   */
  public void destroyApp(boolean cond) {
    System.out.println("ThreadDemo: destroyApp(" + cond + ")");
    System.out.println("ThreadDemo: Stopping threads from outsdide");
    runThreads = false;
    try {
      System.out.println("ThreadDemo: Waiting for threads to die");
      thread1.join();
      thread2.join();
    } catch(InterruptedException e) {
      System.out.println(e);
    }
    System.out.println("ThreadDemo: All threads died");
  }
}
```

13 Differences from the TC45

For those who are familiar with the Siemens WM TC45 product this is an overview of the main differences between TC45 and TC65.

- “real” TCP and UDP access interfaces: SocketConnection, ServerSocketConnection, UDPDatagramConnection.
  Use of StreamConnection, StreamConnectionNotifier, DatagramConnection is now discouraged.
- Serial interfaces are swapped: Standard.out is on ASC1, CommConnection on ASC0 (->open COM0 instead of COM1)
- No IO pin multiplexing: GPIOs, DAI and the serial interface for CommConnection do not share any pins, so the selection mechanism no longer exists.
- The CommConnection interface which used to be proprietary (com.siemens.icm.io) is now part of the standard package (javax.microedition.io).
- No interface emulation on the PC: When running a MIDlet under the emulator, it is completely executed in the connected module and therefore uses the modules “real” interfaces. The emulation of interfaces such as networking, file system or serial interface on the PC side no longer exists.
- .jad files required: A suitable descriptor file is now not only required for OTAP but in all cases. An absent or invalid .jad file causes an error when starting an application (at^sjra or autostart).
- Mandatory attributes: the attributes MicroEdition-Profile and MicroEdition-Configuration are now mandatory attributes in the manifest and .jad file.
- Flexible echo: When using the ATCommand class the “echo” is can be switched on and off as in non-Java mode. The default is echo on. In the TC45 the echo was always off.
- activeCount result: The method activeCount of class Thread returns the number of threads currently present in the VM. In the TC45 only the non sleeping or waiting threads were counted.